node-wiki-book

y2468101216

Published
with GitBook



目錄

關於本書

這是一本關於 Node.js 技術的開放源碼電子書，我們使用 GitBook 維護電子書內容，並交由 GitBook 自動線上發佈。本書提供 PDF、EPUB、MOBI 及 HTML 等格式，您除了可以在網站檢視本書所有內容，也可以將電子書下載至閱讀器保存。

本書的線上閱讀網址，與 GitBook 資料同步更新。

https://www.gitbook.com/read/book/y2468101216/node-wiki-book

如果您想要取得本書的其他格式，可以從 GitBook 的電子書專頁下載最新版本。

https://www.gitbook.com/book/y2468101216/node-wiki-book/details

本書適合 Node.js 初學者至進階開發者，也歡迎您在學習時一起參與本書內容撰寫。

有興趣者請來信 y2468101216@gmail.com

編寫語法與規範

請參考連結，

```
http://sphinx-doc.org/rest.html
```

根目錄結構

- SUMMARY.md -> 本書編排列表
- cover.jpg -> 本書封面(徵求封面設計者)
- LINKS.md -> 外部連結
- README.md -> 本書說明
- zh-tw -> 內含各章節詳細資料，編排方式可參考 SUMMARY.md
- src -> 範例程式碼擺放位置(目前全部嵌在網頁裡面)
- APPENDIX.md -> 附錄

授權

作者

本書由 Node.js Taiwan 社群成員協作，以下名單依照字母排序。

- Caesar Chi (clonn)
- Dca
- Fillano Feng (fillano)
- Hsu Ping Feng
- Kevin Shu (Kevin)
- lyhcode http://about.me/lyhcode
- MarkLin
- Yun
- Zack

Node.js Taiwan 是一個自由開放的技術學習社群，我們歡迎您加入一起學習、研究及分享。

下載電子書

線上閱讀本書。

https://www.gitbook.com/read/book/y2468101216/node-wiki-book

PDF 格式，適合一般電腦及7吋以上平板電腦閱讀

https://www.gitbook.com/download/pdf/book/y2468101216/node-wiki-book

EPUB 格式，適合 iPad、iPhone 行動裝置閱讀

https://www.gitbook.com/download/epub/book/y2468101216/node-wiki-book

MOBI 格式，適合 Kindle 電子書閱讀器

https://www.gitbook.com/download/mobi/book/y2468101216/node-wiki-book

原始碼（目前全部嵌在網頁裡面）

本書最新的原始碼（中文版）網址如下：

https://github.com/y2468101216/node-wiki-gitbook

精選文章收錄流程

精選文章的用意是鼓勵作者在自已的網誌發表 Node.js 教學，再由 Node.js Taiwan 社群挑選系列文章列入電子書的精選文集。

1. 將文章標題及連結貼到「精選文章」分類
2. 社群工作小組以 E-Mail 通知作者文章列入精選，並邀請將內文授權給 Node.js Taiwan 電子書分享
3. 作者同意後，由工作小組負責整理圖文，發佈至電子書
4. 以 E-Mail 寄出感謝函通知原作者文章已收錄，依作者意願調整文章內容
5. 於 Node.js Taiwan 首頁及粉絲專頁推薦作者的文章

前言

Node.js 是 JavaScript 程式語言的開發框架，由 0.4.x 至目前 v4.0.0 版本核心上已經有許多變更，當然目的只有一個就是讓開發變得更簡單，速度能夠更快，這是所有人奮鬥的目標。

而 Node.js 華文維基平台，主力由 Node.js 一群喜好 JavaScript 開發者共同主筆，內文以中文為主，希望能夠降低開發者學習門檻，藉由我們拋磚引玉，讓更多華人開發者共同學習、討論。

Node.js 簡介

Node.js 是一個高效能、易擴充的網站應用程式開發框架 (Web Application Framework)。它誕生的原因，是為了讓開發者能夠更容易開發高延展性的網路服務，不需要經過太多複雜的調校、效能調整及程式修改，就能滿足網路服務在不同發展階段對效能的要求。

Ryan Dahl 是 Node.js 的催生者，目前任職於 Joyent 主機託管服務公司。他開發 Node.js 的目的，就是希望能解決 Apache 在連線數量過高時，緩衝區 (buffer) 和系統資源會很快被耗盡的問題，希望能建立一個新的開發框架以解決這個問題。因此嘗試使用效能十分優秀的 V8 JavaScript Engine，讓網站開發人員使用熟悉的 JavaScript 語言，也能應用於後端服務程式的開發，並且具有出色的執行效能。

JavaScript 是功能強大的物件導向程式語言，但是在 JavaScript 的官方規格中，主要是定義網頁 (以瀏覽器為基礎) 應用程式需要的應用程式介面 (API)，對應用範圍有所侷限。為使 JavaScript 能夠在更多用途發展， CommonJS 規範一組標準函式庫 (standard library)，使 JavaScript 的應用範圍能夠和 Ruby 、 Python 及 Java 等語言同樣豐富，並且能在不同的 CommonJS 兼容 (compliant) JavaScript 執行環境中，使程式碼具有可攜性。

瀏覽器的 JavaScript 與實現 CommonJS 規範的 Node.js 有何不同呢？瀏覽器的 JavaScript 提供 XMLHttpRequest ，讓程式可以和網頁伺服器建立資料傳輸連線，但這通常只能適用於網站開發的需求，因為我們只能用 XMLHttpRequest 與網頁伺服器通訊，卻無法利用它建立其他類型如 Telnet / FTP / NTP 的伺服器通訊。如果我們想開發網路服務程式，例如 SMTP 電子郵件伺服器，就必須使用 Sockets 建立 TCP (某些服務則用 UDP) 監聽及連線，其他程式語言如 PHP 、 Java 、 Python 、 Perl 及 Ruby 等，在標準開發環境中皆有提供 Sockets API ，而瀏覽器的 JavaScript 基於安全及貼近網站設計需求的考量下，並未將 Sockets 列入標準函式庫之中。而 CommonJS 的規範就填補了這種基礎函式庫功能的空缺，遵循 CommonJS 規範的 Node.js 可以直接使用 Sockets API 建立各種網路服務程式，也能夠讓更多同好基於 JavaScript 開發符合 Node.js 的外掛模組 (Module)。

開發人員所編寫出來的 JavaScript 腳本程式，怎麼可能會比其他語言寫出來的網路程式還要快上許多呢？以前的網路程式原理是將使用者每次的連線 (connection) 都開啟一個執行緒 (thread)，當連線爆增的時候將會快速耗盡系統效能，並且容易產生阻塞 (block)。

Node.js 對於資源的調配有所不同，當程式接收到一筆連線 (connection)，會通知作業系統透過 epoll, kqueue, /dev/poll 或 select 將連線保留，並且放入 heap 中配置，先讓連線進入休眠 (sleep) 狀態，當系統通知時才會觸發連線的 callback。這種處理連線方式只會佔用掉記憶體，並不會使用到 CPU 資源。另外因為採用 JavaScript 語言的特性，每個 request 都會有一個 callback，如此可以避免發生 block。

基於 callback 特性，目前 Node.js 大多應用於 Comet(long polling) Request Server，或者是高連線數量的網路服務上，目前也有許多公司將 Node.js 設為內部核心網路服務之一。在 Node.js 也提供了外掛管理 (Node package management)，讓愛好 Node.js 輕易開發更多有趣的服務、外掛，並且提供到 npm 讓全世界使用者快速安裝使用。

本書最後執行測試版本為Node.js v4.0.0，相關 API 文件可查詢 https://nodejs.org

本書所有範例均可於 Linux, Windows 上執行，如遇到任何問題歡迎至 http://nodejs.tw ，詢問對於 Node.js 相關問題，或者可以加入 FB 社團 Node.js台灣

附錄 Node.js 與 JavaScript

JavaScript 基本型態

JavaScript 有以下幾種基本型態。

- Boolean
- Number
- String
- null
- undefined

變數宣告的方式，就是使用 var，結尾使用『;』，如果需要連續宣告變數，可以使用 『,』 做為連結符號。

```
// 宣告 x 為 123, 數字型態
var x=123;

// 宣告 a 為456, b 為 'abc' 字串型態
var a=456,
    b='abc';
```

布林值

布林，就只有兩種數值, true, false

```
var a=true,
    b=false;
```

數字型別

Number 數字型別，可以分為整數，浮點數兩種，

```
var a=123,
    b=123.456;
```

字串型別

字串，可以是一個字，或者是一連串的字，可以使用 " 或 "" 做為字串的值。 (盡量使用雙引號來表達字串，因為在node裡不會把單引號框住的文字當作字串解讀)

```
var a="a",
    a='abc';
```

運算子

基本介紹就是 +, -, *, / 邏輯運算就是 && (and), || (or), ^ (xor), 比較式就是 >, <, !=, !==, ==, ===, >=, <=

判斷式

這邊突然離題，加入判斷式來插花，判斷就是 if，整個架構就是，

```
if (判斷a) {
  // 判斷a 成立的話，執行此區域指令
} else if (判斷b) {
  // 判斷a 不成立，但是 判斷b 成立，執行此區域指令
} else {
  // 其餘的事情在這邊處理
}
```

整體架構就如上面描述，非 a 即 b的狀態，會掉進去任何一個區域裡面。整體的判斷能夠成立，只要判斷轉型成 Boolean 之後為 true，就會成立。大家可以這樣子測試，

```
Boolean(判斷);
```

應用

會突然講 if 判斷式，因為，前面有提到 Number, String 兩種型態，但是如果我們測試一下，新增一個 test.js

```
var a=123,
    b='123';

if (a == b) {
  console.log('ok');
}
```

編輯 test.js 完成之後，執行底下指令

```
node test.js
// print: ok
```

輸出結果為 ok。

這個結果是有點迴異， a 為 Number, b 為 String 型態，兩者相比較，應該是為 false 才對，到底發生什麼事情？這其中原因是，在判斷式中使用了 == ，JavaScript 編譯器，會自動去轉換變數型態，再進行比對，因此 a == b 就會成立，如果不希望轉型產生，就必須要使用 === 做為判斷。

```
if (a === b) {
  console.log('ok');
} else {
  console.log('not ok');
}
// print: not ok
```

轉型

如果今天需要將字串，轉換成 Number 的時候，可以使用 parseInt, parseFloat 的方法來進行轉換，

```
var a='123';
console.log(typeof parseInt(a, 10));
```

使用 typeof 方法取得資料經過轉換後的結果，會取得，

```
number
```

要注意的是，記得 parseInt 後面要加上進位符號，以免造成遺憾，在這邊使用的是 10 進位。

Null & undefined 型態差異

空無是一種很奇妙的狀態，在 JavaScript 裡面，null, undefined 是一種奇妙的東西。今天來探討什麼是 null ，什麼是 undefined.

null

變數要經過宣告，賦予 null ，才會形成 null 型態。

```
var a=null;
```

null 在 JavaScript 中表示一個空值。

undefined

從字面上就表示目前未定義，只要一個變數在初始的時候未給予任何值的時候，就會產生 undefined

```
var a;

console.log(a);

// print : undefined
```

這個時候 a 就是屬於 undefined 的狀態。 另外一種狀況就是當 Object 被刪除的時候。

```
var a = {};
delete a;
console.log(a);

//print: undefined.
```

Object 在之後會介紹，先記住有這個東西。而使用 delete 的時候，就可以讓這個 Object 被刪除，就會得到結果為 undefined.

兩者比較

null, undefined 在本質上差異並不大，不過實質上兩者並不同，如果硬是要比較，建議使用 === 來做為判斷標準，避免 null, undefined 這兩者被強制轉型。

```
var a=null,
    b;

if (a === b) {
  console.log('same');
} else {
  console.log('different');
}

//print: different
```

從 typeof 也可以看到兩者本質上的差異，

```
typeof null;
//print: 'object'

typeof undefined;
//print: 'undefined'
```

null 本質上是屬於 object, 而 undefined 本質上屬於 undefined， 意味著在 undefined 的狀態下，都是屬於未定義。

如果用判斷式來決定，會發現另外一種狀態

```
Boolean(null);
// false

Boolean(undefined);
// false
```

可以觀察到，如果一個變數值為 null, undefined 的狀態下，都是屬於 false。

這樣說明應該幫助到大家了解，其實要判斷一個物件、屬性是否存在，只需要使用 if

```
var a;

if (!a) {
  console.log('a is not existed');
}

//print: a is not existed
```

a 為 undefined 由判斷式來決定，是屬於 False 的狀態。

JavaScript Array

陣列也是屬於 JavaScript 的原生物件之一，在實際開發會有許多時候需要使用 Array 的方法，先來介紹一下陣列要怎麼宣告。

陣列宣告

宣告方式,

```
var a=['a', 'b', 'c'];

var a=new Array('a', 'b', 'c');
```

以上這兩種方式都可以宣告成陣列,接著我們將 a 這個變數印出來看一下,

```
console.log(a);
//print: [0, 1, 2]
```

Array 的排列指標從 0 開始,像上面的例子來說, a 的指標就有三個,0, 1, 2,如果要印出特定的某個陣列數值,使用方法,

```
console.log(a[1]);
//print: b
```

如果要判斷一個變數是不是 Array 最簡單的方式就是直接使用 Array 的原生方法,

```
var a=['a', 'b', 'c'];

console.log(Array.isArray(a));
//print: true

var b='a';
console.log(Array.isArray(b));
//print: false
```

如果要取得陣列變數的長度可以直接使用,

```
console.log(a.length);
```

length 為一個常數,型態為 Number,會列出目前陣列的長度。

pop, shift

以前面所宣告的陣列為範例,

```
var a=['a', 'b', 'c'];
```

使用 pop 可以從最後面取出陣列的最後一個值。

```
console.log(a.pop());
//print: c

console.log(a.length);
//print: 2
```

同時也可以注意到，使用 pop 這個方法之後，陣列的數值也會被輸出。另外一個跟 pop 很像的方式就是 shift,

```
console.log(a.shift());
//print: a

console.log(a.length);
//print: 1
```

shift 跟 pop 最大的差異，就是從最前面將數值取出，同時也會讓呼叫的陣列少一個數值。

slice

前面提到 pop, shift 就不得不說一下 slice，使用方式,

```
console.log(a.slice(1,3));
//print: 'b', 'c'
```

第一個參數為起始指標，第二個參數為結束指標，會將這個陣列進行切割，變成一個新的陣列型態。 如果需要給予新的變數，就可以這樣子做，完整的範例。

```
var a=['a', 'b', 'c'];

var b=a.slice(1,3);

console.log(b);
//print: 'b', 'c'
```

concat

concat 這個方法，可以將兩個 Array 組合起來，

```
var a=['a'];

var b=['b', 'c'];

console.log(a.concat(b));
//print: 'a', 'b', 'c'
```

concat 會將陣列組合，之後變成全新的數組，如果以例子來說，a 陣列希望變成 ['a', 'b', 'c']，可以重新將數值分配給 a，範例來說

```
a = a.concat(b);
```

Iterator

陣列資料，必須要有 Iterator，將資料巡迴一次，通常是使用迴圈的方式，

```
var a=['a', 'b', 'c'];

for(var i=0; i < a.length; i++) {
    console.log(a[i]);
}

//print: a
//       b
//       c
```

事實上可以用更簡單的方式進行，

```
var a=['a', 'b', 'c'];

a.forEach(function (val, idx) {
  console.log(val, idx);
});

/*
print:
a, 0
b, 1
c, 2
*/
```

在 Array 裡面可以使用 foreach 的方式進行 iterator， 裡面給予的 function (匿名函式)， 第一個變數為 Array 的 Value, 第二個變數為 Array 的指標。

其實使用 JavaScript 在網頁端與伺服器端的差距並不大，但是為了使 Node.js 可以發揮他最強大的能力，有一些知識還是必要的，所以還是針對這些主題介紹一下。

其中 Event Loop、Scope 以及 Callback 其實是比較需要了解的基本知識， cps、currying、flow control是更進階的技巧與應用。

Event Loop

可能很多人在寫JavaScript時，並不知道他是怎麼被執行的。這個時候可以參考一下jQuery作者John Resig一篇好文章，介紹事件及timer怎麼在瀏覽器中執行：How JavaScript Timers Work。通常在網頁中，所有的JavaScript執行完畢後（這部份全部都在global scope跑，除非執行函數），接下來就是如John Resig解釋的這樣，所有的事件處理函數，以及timer執行的函數，會排在一個queue結構中，利用一個無窮迴圈，不斷從queue中取出函數來執行。這個就是event loop。

（除了John Resig的那篇文章，Nicholas C. Zakas的 "Professional JavaScript for Web Developer 2nd edition"， 在 598 頁剛好也有簡短的說明）

所以在JavaScript中，雖然有非同步，但是他並不是使用執行緒。所有的事件或是非同步執行的函數，都是在同一個執行緒中，利用event loop的方式在執行。至於一些比較慢的動作例如I/O、網頁render, reflow等，實際動作會在其他執行緒跑，等到有結果時才利用事件來觸發處理函數來處理。這樣的模型有幾個好處： 沒有執行緒的額外成本，所以反應速度很快 不會有任何程式同時用到同一個變數，不必考慮lock，也不會產生dead lock 所以程式撰寫很簡單 但是也有一些潛在問題： 任一個函數執行時間較長，都會讓其他函數更慢執行（因為一個跑完才會跑另一個） 在多核心硬體普遍的現在，無法用單一的應用程式instance發揮所有的硬體能力 用Node.js撰寫伺服器程式，碰到的也是一樣的狀況。要讓系統發揮event loop的效能，就要盡量利用事件的方式來組織程式架構。另外，對於一些有可能較為耗時的操作，可以考慮使用 process.nextTick 函數來讓他以非同步的方式執行，避免在同一個函數中執行太久，擋住所有函數的執行。

如果想要測試event loop怎樣在「瀏覽器」中運行，可以在函數中呼叫alert()，這樣會讓所有JavaScript的執行停下來，尤其會干擾所有使用timer的函數執行。有一個簡單的例子，這是一個會依照設定的時間間隔嚴格執行動作的動畫，如果時間過了就會跳過要執行的動作。點按圖片以後，人物會快速旋轉，但是在旋轉執行完畢前按下「delay」按鈕，讓alert訊息等久一點，接下來的動畫就完全不會出現了。

Scope 與 Closure

要快速理解 JavaScript 的 Scope（變數作用範圍）原理，只要記住他是Lexical Scope就差不多了。簡單地說，變數作用範圍是依照程式定義時（或者叫做程式文本？）的上下文決定，而不是執行時的上下文決定。

為了維護程式執行時所依賴的變數，即使執行時程式運行在原本的scope之外，他的變數作用範圍仍然維持不變。這時程式依賴的自由變數（定義時不是local的，而是在上一層scope定義的變數）一樣可以使用，就好像被關閉起來，所以叫做Closure。用程式看比較好懂：

```
function outter(arg1) {
  //arg1及free_variable1對inner函數來說，都是自由變數
  var free_variable1 = 3;
  return function inner(arg2) {
    var local_variable1 =2;//arg2及local_variable1對inner函數來說，都
    return arg1 + arg2 + free_variable1 + local_variable1;
  };
}
```

var a = outter(1);//變數a 就是outter函數執行後返回的inner函數

var b = a(4);//執行inner函數，執行時上下文已經在outter函數之外，但是仍然能正常執行，而且可以使用定義在outter函數裡面的arg1及free_variable1變數

console.log(b);//結果10

在JavaScript中，scope最主要的單位是函數（另外有global及eval），所以有可能製造出closure的狀況，通常在形式上都是有巢狀的函數定義，而且內側的函數使用到定義在外側函數裡面的變數。

Closure有可能會造成記憶體洩漏，主要是因為被參考的變數無法被垃圾收集機制處理，造成佔用的資源無法釋放，所以使用上必須考慮清楚，不要造成意外的記憶體洩漏。（在上面的例子中，如果a一直未執行，使用到的記憶體就不會被釋放）

跟透過函數的參數把變數傳給函數比較起來，JavaScript Engine會比較難對Closure進行最佳化。如果有效能上的考量，這一點也需要注意。

Callback

要介紹 Callback 之前， 要先提到 JavaScript 的特色。

JavaScript 是一種函數式語言（functional language），所有JavaScript語言內的函數，都是高階函數(higher order function，這是數學名詞，計算機用語好像是first class function，意指函數使用沒有任何限制，與其他物件一樣)。也就是說，函數可以作為函數的參數傳給函數，也可以當作函數的返回值。這個特性，讓JavaScript的函數，使用上非常有彈性，而且功能強大。

callback在形式上，其實就是把函數傳給函數，然後在適當的時機呼叫傳入的函數。JavaScript使用的事件系統，通常就是使用這種形式。Node.js中，有一個物件叫做EventEmitter，這是Node.js事件處理的核心物件，所有會使用事件處理的函數，都會「繼承」這個物件。（這裡說的繼承，實作上應該像是mixin）他的使用很簡單： 可以使用 物件.on(事件名稱, callback函數) 或是 物件.addListener(事件名稱, callback函數) 把你想要處理事件的函數傳入 在 物件 中，可以使用 物件.emit(事件名稱, 參數...) 呼叫傳入的callback函數 這是Observer Pattern的簡單實作，而且跟在網頁中使用DOM的addEventListener使用上很類似，也很容易上手。不過Node.js是大量使用非同步方式執行的應用，所以程式邏輯幾乎都是寫在callback函數中，當邏輯比較複雜時，大量的callback會讓程式看起來很複雜，也比較難單元測試。舉例來說：

```
var p_client = new Db('integration_tests_20', new Server("127.0.0.1
p_client.open(function(err, p_client) {
  p_client.dropDatabase(function(err, done) {
    p_client.createCollection('test_custom_key', function(err, coll
      collection.insert({'a':1}, function(err, docs) {
        collection.find({'_id':new ObjectID("aaaaaaaaaaaa")}, funct
          cursor.toArray(function(err, items) {
            test.assertEquals(1, items.length);
            p_client.close();
          });
        });
      });
    });
  });
});
```

這是在網路上看到的一段操作mongodb的程式碼，為了循序操作，所以必須在一個callback裡面呼叫下一個動作要使用的函數，這個函數裡面還是會使用callback，最後就形成一個非常深的巢狀。

這樣的程式碼，會比較難進行單元測試。有一個簡單的解決方式，是盡量不要使用匿名函數來當作callback或是event handler。透過這樣的方式，就可以對各個handler做單元測試了。例如：

```
var http = require('http');
var tools = {
  cookieParser: function(request, response) {
    if(request.headers['Cookie']) {
    //do parsing
    }
  }
};
var server = http.createServer(function(request, response) {
  this.emit('init', request, response);
  //...
});
server.on('init', tools.cookieParser);
server.listen(8080, '127.0.0.1');
```

更進一步，可以把tools改成外部module，例如叫做tools.js：

```
module.exports = {
  cookieParser: function(request, response) {
    if(request.headers['Cookie']) {
    //do parsing
    }
  }
};
```

然後把程式改成：

```
var http = require('http');

var server = http.createServer(function(request, response) {
  this.emit('init', request, response);
  //...
});
server.on('init', require('./tools').cookieParser);
server.listen(8080, '127.0.0.1');
```

這樣就可以單元測試cookieParser了。例如使用nodeunit時，可以這樣寫：

```
var testCase = require('nodeunit').testCase;
module.exports = testCase({
  "setUp": function(cb) {
    this.request = {
      headers: {
      Cookie: 'name1:val1; name2:val2'
    }
  };
  this.response = {};
  this.result = {name1:'val1',name2:'val2'};
    cb();
  },
  "tearDown": function(cb) {
    cb();
  },
  "normal_case": function(test) {
    test.expect(1);
    var obj = require('./tools').cookieParser(this.request, this.re
    test.deepEqual(obj, this.result);
    test.done();
  }
});
```

善於利用模組，可以讓程式更好維護與測試。

CPS（Continuation-Passing Style）

cps是callback使用上的特例，形式上就是在函數最後呼叫callback，這樣就好像把函數執行後把結果交給callback繼續運行，所以稱作continuation-passing style。利用cps，可以在非同步執行的情況下，透過傳給callback的這個cps callback來獲知callback執行完畢，或是取得執行結果。例如：

```
<html>
  <body>
    <div id="panel" style="visibility:hidden"></div>
  </body>
</html>

<script>
  var request = new XMLHttpRequest();
  request.open('GET', 'test749.txt?timestamp='+new Date().getTime()
  request.addEventListener('readystatechange', function(next){
    return function() {
      if(this.readyState===4&&this.status===200) {
        next(this.responseText);//<==傳入的cps callback在動作完成時執行
      }
    };
  }(function(str){//<==這個匿名函數就是cps callback
    document.getElementById('panel').innerHTML=str;
    document.getElementById('panel').style.visibility = 'visible';
  }), false);
  request.send();
</script>
```

進一步的應用，也可以參考2-6 流程控制。

函數返回函數與Currying

前面的cps範例裡面，使用了函數返回函數，這是為了把cps callback傳遞給onreadystatechange事件處理函數的方法。（因為這個事件處理函數並沒有設計好會傳送/接收這樣的參數）實際會執行的事件處理函數其實是內層返回的那個函數，

之外包覆的這個函數，主要是為了利用Closure，把next傳給內層的事件處理函數。這個方法更常使用的地方，是為了解決一些scope問題。例如：

```
<script>
var accu=0,count=10;
for(var i=0; i<count; i++) {
  setTimeout(
    function(){
      count--;
      accu+=i;
      if(count<=0)
        console.log(accu)
    }
  , 50)
}
</script>
```

最後得出的結果會是100，而不是想像中的45，這是因為等到setTimeout指定的函數執行時，變數i已經變成10而離開迴圈了。要解決這個問題，就需要透過Closure來保存變數i：

```
<script>
var accu=0,count=10;
for(var i=0; i<count; i++) {
  setTimeout(
    function(i) {
     return function(){
     count--;
       accu+=i;
       if(count<=0)
         console.log(accu)
     };
   }(i)
  , 50)
}
//淺藍色底色的部份，是跟上面例子不一樣的地方
</script>
```

函數返回函數的另外一個用途，是可以暫緩函數執行。例如：

```
function add(m, n) {
  return m+n;
}
var a = add(20, 10);
console.log(a);
```

add這個函數，必須同時輸入兩個參數，才有辦法執行。如果我希望這個函數可以先給它一個參數，等一些處理過後再給一個參數，然後得到結果，就必須用函數返回函數的方式做修改：

```
function add(m) {
  return function(n) {
    return m+n;
  };
}
var wait_another_arg = add(20);//先給一個參數
var a = function(arr) {
  var ret=0;
  for(var i=0;i<arr.length;i++) ret+=arr[i];
  return ret;
}([1,2,3,4]);//計算一下另一個參數
var b = wait_another_arg(a);//然後再繼續執行
console.log(b);
```

像這樣利用函數返回函數，使得原本接受多個參數的函數，可以一次接受一個參數，直到參數接收完成才執行得到結果的方式，有一個學名就叫做...Currying

綜合以上許多奇技淫巧，就可以透過用函數來處理函數的方式，調整程式流程。接下來看看...

流程控制

（以sync方式使用async函數、避開巢狀callback循序呼叫async callback等奇技淫巧）

建議參考：

- http://howtonode.org/control-flow
- http://howtonode.org/control-flow-part-ii
- http://howtonode.org/control-flow-part-iii
- http://blog.mixu.net/2011/02/02/essential-node-js-patterns-and-snippets

這幾篇都是非常經典的Node.js/JavaScript流程控制好文章（阿，mixu是在介紹一些pattern時提到這方面的主題）。不過我還是用幾個簡單的程式介紹一下做法跟概念：

並發與等待

下面的程式參考了mixu文章中的做法：

```
var wait = function(callbacks, done) {
  console.log('wait start');
  var counter = callbacks.length;
  var results = [];
  var next = function(result) {//接收函數執行結果，並判斷是否結束執行
    results.push(result);
    if(--counter == 0) {
      done(results);//如果結束執行，就把所有執行結果傳給指定的callback處理
    }
  };
  for(var i = 0; i < callbacks.length; i++) {//依次呼叫所有要執行的函數
    callbacks[i](next);
  }
  console.log('wait end');
}

wait(
  [
    function(next){
      setTimeout(function(){
        console.log('done a');
        var result = 500;
        next(result)
      },500);
```

```
    },
    function(next){
      setTimeout(function(){
        console.log('done b');
        var result = 1000;
        next(result)
      },1000);
    },
    function(next){
      setTimeout(function(){
        console.log('done c');
        var result = 1500;
        next(1500)
      },1500);
    }
  ],
  function(results){
    var ret = 0, i=0;
    for(; i<results.length; i++) {
      ret += results[i];
    }
    console.log('done all. result: '+ret);
  }
);
```

執行結果：

```
wait start
wait end
done a
done b
done c
done all. result: 3000
```

可以看出來，其實wait並不是真的等到所有函數執行完才結束執行，而是在所有傳給他的函數執行完畢後（不論同步、非同步），才執行處理結果的函數（也就是done()）

不過這樣的寫法，還不夠實用，因為沒辦法實際讓函數可以等待執行完畢，又能當作事件處理函數來實際使用。上面參考到的Tim Caswell的文章，裡面有一種解法，不過還需要額外包裝（在他的例子中）Node.js核心的fs物件，把一些函數（例如readFile）用Currying處理。類似像這樣：

```
var fs = require('fs');
var readFile = function(path) {
  return function(callback, errback) {
    fs.readFile(path, function(err, data) {
      if(err) {
        errback();
      } else {
        callback(data);
      }
    });
  };
}
```

其他部份可以參考Tim Caswell的文章，他的Do.parallel跟上面的wait差不多意思，這裡只提示一下他沒說到的地方。

另外一種做法是去修飾一下callback，當他作為事件處理函數執行後，再用cps的方式取得結果：

```
<script>
function Wait(fns, done) {
  var count = 0;
  var results = [];
  this.getCallback = function(index) {
    count++;
    return (function(waitback) {
      return function() {
        var i=0,args=[];
        for(;i<arguments.length;i++) {
          args.push(arguments[i]);
        }
        args.push(waitback);
        fns[index].apply(this, args);
      };
```

```
    })(function(result) {
      results.push(result);
      if(--count == 0) {
        done(results);
      }
    });
  }
}
var a = new Wait(
  [
    function(waitback){
      console.log('done a');
      var result = 500;
      waitback(result)
    },
    function(waitback){
      console.log('done b');
      var result = 1000;
      waitback(result)
    },
    function(waitback){
      console.log('done c');
      var result = 1500;
      waitback(result)
    }
  ],
  function(results){
    var ret = 0, i=0;
    for(; i<results.length; i++) {
      ret += results[i];
    }
    console.log('done all. result: '+ret);
  }
);
var callbacks = [a.getCallback(0),a.getCallback(1),a.getCallback(0)

//一次取出要使用的callbacks，避免結果提早送出
setTimeout(callbacks[0], 500);
setTimeout(callbacks[1], 1000);
setTimeout(callbacks[2], 1500);
```

```
setTimeout(callbacks[3], 2000);
//當所有取出的callbacks執行完畢，就呼叫done()來處理結果
</script>
```

執行結果：

```
done a
done b
done a
done c
done all. result: 3500
```

上面只是一些小實驗，更成熟的作品是Tim Caswell的step：https://github.com/creationix/step

如果希望真正使用同步的方式寫非同步，則需要使用Promise.js這一類的library來轉換非同步函數，不過他結構比較複雜XD（見仁見智，不過有些人認為Promise有點過頭了）：http://blogs.msdn.com/b/rbuckton/archive/2011/08/15/promise-js-2-0-promise-framework-for-javascript.aspx

如果想不透過其他Library做轉換，又能直接用同步方式執行非同步函數，大概就要使用一些需要額外compile原始程式碼的方法了。例如Bruno Jouhier的streamline.js：https://github.com/Sage/streamlinejs

循序執行

循序執行可以協助把非常深的巢狀callback結構攤平，例如用這樣的簡單模組來做（serial.js）：

```
module.exports = function(funs) {
  var c = 0;
  if(!isArrayOfFunctions(funs)) {
    throw('Argument type was not matched. Should be array of funct
  }
  return function() {
    var args = Array.prototype.slice.call(arguments, 0);
    if(!(c>=funs.length)) {
      c++;
      return funs[c-1].apply(this, args);
    }
  };
}

function isArrayOfFunctions(f) {
  if(typeof f !== 'object') return false;
  if(!f.length) return false;
  if(!f.concat) return false;
  if(!f.splice) return false;
  var i = 0;
  for(; i<f.length; i++) {
    if(typeof f[i] !== 'function') return false;
  }
  return true;
}
```

簡單的測試範例（testSerial.js），使用fs模組，確定某個path是檔案，然後讀取印出檔案內容。這樣會用到兩層的callback，所以測試中有使用serial的版本與nested callbacks的版本做對照：

```
var serial = require('./serial'),
  fs = require('fs'),
  path = './dclient.js',
  cb = serial([
  function(err, data) {
    if(!err) {
      if(data.isFile) {
        fs.readFile(path, cb);
```

```
      }
    } else {
      console.log(err);
    }
  },
  function(err, data) {
    if(!err) {
      console.log('[flattened by searial:]');
      console.log(data.toString('utf8'));
    } else {
      console.log(err);
    }
  }
]);
fs.stat(path, cb);

fs.stat(path, function(err, data) {
  //第一層callback
  if(!err) {
    if(data.isFile) {
      fs.readFile(path, function(err, data) {
        //第二層callback
        if(!err) {
          console.log('[nested callbacks:]');
          console.log(data.toString('utf8'));
        } else {
          console.log(err);
        }
      });
    } else {
      console.log(err);
    }
  }
});
```

關鍵在於，這些callback的執行是有順序性的，所以利用serial返回的一個函數cb來取代這些callback，然後在cb中控制每次會循序呼叫的函數，就可以把巢狀的callback攤平成循序的function陣列（就是傳給serial函數的參數）。

測試中的./dclient.js是一個簡單的dnode測試程式，放在跟testSerial.js同一個目錄：

```
var dnode = require('dnode');

dnode.connect(8000, 'localhost',  function(remote) {
  remote.restart(function(str) {
    console.log(str);
    process.exit();
  });
});
```

執行測試程式後，出現結果：

[flattened by searial:]

```
var dnode = require('dnode');

dnode.connect(8000, 'localhost',  function(remote) {
  remote.restart(function(str) {
    console.log(str);
    process.exit();
  });
});
```

[nested callbacks:]

```
var dnode = require('dnode');

dnode.connect(8000, 'localhost',  function(remote) {
  remote.restart(function(str) {
    console.log(str);
    process.exit();
  });
});
```

對照起來看，兩種寫法的結果其實是一樣的，但是利用serial.js，巢狀的callback結構就會消失。

不過這樣也只限於順序單純的狀況，如果函數執行的順序比較複雜（不只是一直線），還是需要用功能更完整的流程控制模組比較好，例如 https://github.com/caolan/async 。

Node.js 安裝與設定

本篇將講解如何在各個不同OS建立NodeJS 環境，目前NodeJS在不同作業系統中可以直接使用指令快速架設。以下各不同作業系統解說如何安裝NodeJS與nvm。

What's nvm

nvm是一種Node.js的版本管理工具，在現在Node.js已經分成了stable跟develop，適時的切換版本是必須的。因為你有可能需要預先開發develop上的功能，但還是要維護stable上的bug， 且你使用nvm切換版本時npm也會一併安裝或切換。

Linux

在這裡我們不使用手動安裝，而是使用nvm自帶的install script

可以使用curl

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.29.0/i
```

或者Wget

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.29.0/
```

來執行這些install script，安裝完以後請重新連結你的vps或者重開你的server

手動安裝請依照參考資料裡nvm的github尋找Manual install的標題依序做下去(不推薦)

安裝完成後可以打

```
nvm
```

會顯示下面圖片：

```
Node Version Manager

Note: <version> refers to any version-like string nvm understands. This includes
:
  - full or partial version numbers, starting with an optional "v" (0.10, v0.1.2
, v1)
  - default (built-in) aliases: node, stable, unstable, iojs, system
  - custom aliases you define with `nvm alias foo`

Usage:
  nvm help                                    Show this message
  nvm --version                               Print out the latest released versio
n of nvm
  nvm install [-s] <version>                  Download and install a <version>, [-
s] from source. Uses .nvmrc if available
    --reinstall-packages-from=<version>       When installing, reinstall packages
installed in <node|iojs|node version number>
  nvm uninstall <version>                     Uninstall a version
  nvm use [--silent] <version>                Modify PATH to use <version>. Uses .
nvmrc if available
  nvm exec [--silent] <version> [<command>]   Run <command> on <version>. Uses .nv
```

OSX

OSX同樣可以用curl or wget的install script來安裝(同上)，homebrew也可以，但我並不推薦，所以不再撰述。

Windows

windows上並不支援nvm，但是有其他人開發了如nvmw、nvm-windows等來支援windows，這邊我們選擇star數比較多的nvm-windows來講解

installer link:https://github.com/coreybutler/nvm-windows/releases 選擇最新版nvm-setup.zip進行下載，解壓縮開了以後選擇nvm-setup.exe進行安裝，遵循指示下一步即可。

注意更新的時候只要重複安裝的步驟，安裝目錄選擇一樣就好。

nvm指令簡介

這些nvm的指令皆可在https://github.com/creationix/nvm裡找到

- install

```
nvm install 4.2.1
nvm install stable
```

這個指令是使用Node.js必須先做的，因為在安裝完nvm後Node.js並沒有被安裝。你可以看到他可以帶兩種參數，一個是版本號、一個是stable or unstable。

版本號除了第一位必填以外，其他都選填，沒填的情況下會自動抓該版本下最新的。 EX:nvm install 4會安裝4.2.1

stable or unstable一個是安裝穩定版、一個是安裝最新版，不過就現在而言兩者是一樣的(2015-10-14)。

- use

```
npm use 4.2.1
npm use stable
```

此指令會切換node的版本，你可以在cmd或者terminal打：

```
node -v
```

查看現在版本，參數的設定同install。

- run

```
nvm run 4.2.1 [node file or node command]
nvm run stable [node file or node command]
```

要求nvm以特定版本運行node file或者node command，在測試ES6跟非ES6的code此功能十分重要。 此功能不會影響你本機裡的版本

EX:假設妳現在系統版本是4.1.1

```
Yuns-MacBook-Pro:~ Yun$ node -v
v4.1.1
Yuns-MacBook-Pro:~ Yun$ nvm run 0.12.7 -v
Running node v0.12.7 (npm v2.14.4)
v0.12.7
Yuns-MacBook-Pro:~ Yun$
```

由圖可以知道run的版本不受到現在系統版本的影響

- ls

```
nvm ls
nvm ls-remote
```

ls可以讓你查看"本地端"安裝的所有版本 ls-remote可以讓你查看，現在官方放出的所有版本。

- 設定node預設使用版本

```
nvm alias default stable
nvm alias default 4.2.1
```

此指令可以讓你設定每次開啟terminal的node預設版本是多少

結語

nvm是每個開發者都需要學習的東西，我建議務必做一遍，而不要從官網上下載檔案來安裝。

參考資料

- nvm:https://github.com/creationix/nvm
- nvm-windows:https://github.com/coreybutler/nvm-windows

前言

有慣常留意 Node.js 社群的開發者們應該都知道剛剛有一個重大更新，現時最新的版本到了 v4.1。然而根據 Node.js 在發佈 v4.0 的時候釋放的官方文檔，指出剛在六月正式發佈 ECMAScript 6 (下稱 ES6) 會分三個階段納入最新的版本當中。它們分別是:

- Shipping features: 已經完成整合並且被 V8 開發團隊視為穩定
- Staged features: 大致完成整合但並不能確定能夠穩定運行
- In progress features 僅用於測試

(註: 在 Node.js v4.0 或更新版本的環境中)

除了 Shipping features 以外，開發者如要使用其他的語法特性需要自行承擔風險。

如何在 **Node.js** 啟用對 **ES6** 的支援

在 **v4.0** 及其以後的更新版本

- Shipping features 並不需要加上 runtime flag 已經可以直接在最新版本的 Node.js 環境中使用
- Staged features 需要加上 runtime flag (`--es_staging` 或 `--harmony`) 才可以使用
- In progress features 需要加上 runtime flag `--harmony_<name>` , 在 harmony_ 後面的是那語法特性的名稱,如果開發者想知道有甚麼是正在整合當中的話,可以使用 `node --v8-options | grep "in progress"` 去查詢

在 v4.0 下的 In progress 特性:

```
--harmony_modules (enable "harmony modules" (in progress))
--harmony_array_includes (enable "harmony Array.prototype.includes'
--harmony_regexps (enable "harmony regular expression extensions" (
--harmony_proxies (enable "harmony proxies" (in progress))
--harmony_sloppy (enable "harmony features in sloppy mode" (in prog
--harmony_unicode_regexps (enable "harmony unicode regexps" (in pro
--harmony_reflect (enable "harmony Reflect API" (in progress))
--harmony_destructuring (enable "harmony destructuring" (in progres
--harmony_sharedarraybuffer (enable "harmony sharedarraybuffer" (in
--harmony_atomics (enable "harmony atomics" (in progress))
--harmony_new_target (enable "harmony new.target" (in progress))
```

在 v4.0 以前的版本

由於在 v4.0 以前的版本並不原生支援 ES6 的語法特性,所以我們需要一個 JavaScript 編譯器,把 ES6 的語法轉換成 ES5 的版本。Babel 是一個開源專案,如果需要在舊的 Node.js 環境中寫 ES6 的語法就可以用到它,使用的方法很容易,先用 npm 把它安裝起來。

```
$npm install babel -g
```

然後就直接可以轉換 ES6 的語法

```
babel myes6.js
```

Shipping Features

以下這些語法特性已經在最新的版本環境釋出:

索引

- let,const
- class
- Map
- WeakMap
- Set
- WeakSet

- Typed Arrays
- Generator
- Binary and Octal
- Object Literal Extension
- New String Methods
- Symbols
- Template Strings
- Arrow Functions
- Promises
- for...of Loops

let, const

ES6 引入了塊級域的變量(block scope variables),使變量的作用域限制於兩個括號裡頭。跟 `var` 不同的是 `var` 所定義的變量要麼是全局(global),要麼是函數域(function scope),不能是塊級域的。比對以下例子就會明白。

```
var globalVar = 1;
if (true) {
  globalVar = 3;
}
console.log(globalVar); // 3
```

若使用 `let` 定義變數的話,在 Block 以外想要知道它的值是不能夠的

```
if (true) {
  let blockVar = 3;
}
console.log(blockVar); // undefined
```

錯誤使用 `let` 所引起的問題可以參照這裡

至於 `const` 固名思義就是常數,是不可變的(immutable)。

```
const constant = 3;
constant = 0;
console.log(constant); // 3
```

同一個名的常數亦不能重覆宣告,否則會引起 `TypeError` 。

```
const constant = 3;
const constant = 3; // TypeError: Identifier 'constant' has already
```

class

事實上這不是一種新加入的面向編程概念,然而這只是把現有 JavaScript 裡頭基於原型(prototype)的繼承(inheritance)做法重新包裝,是一種語法糖(syntax sugar)而已,使程式碼更加簡單易明。看看在 ES6 之前的做法是如何:

```
var Plane = function () {}
Plane.prototype.landing = function () {}
function A380 () {}
A380.prototype = new Plane();
var emirates_a380 = new A380 ();
console.log(emirates_a380 instanceof A380); // true
console.log(emirates_a380 instanceof Plane); // true
```

再看看在 ES6 裡頭使用 `class`

```
'use strict';

class Plane {
  constructor () {
      // ...
  }
  takeoff () {
    console.log('Taking off');
  }
}

class A380 extends Plane {
  constructor () {
    super();
  }
}

var emirates_a380 = new A380();
console.log(emirates_a380 instanceof A380); // true
console.log(emirates_a380 instanceof Plane); // true
```

結果顯而易見,程式碼看起上來更直覺,更清楚易明。

Map

這個物件就是簡單的鍵/值(key/value)對應表,長久以來人們都是使用 Object 來實現 Map 的功能,事實上 ES6 所引入的 Map 還是跟 Object 有所分別:

- 所有 Object 物件的原型都會是 Object 的預設鍵 `Object.prototype` 。可以使用 `map = Object.create(null)` 去創建一個沒有原型的 Object
- Object 的鍵只可以是字符串(String),但 Map 的鍵可以是原始數據 (Primitive) 或 Object。
- Map 的迭代是根據 insertion order,而 Object 的迭代並沒有規範。
- Map 新增了許多額外方法,例如計算有多少對鍵值,從前需要 `Object.keys(myObj).length` ,現在則有 `Map.prototype.size` 。

WeakMap

WeakMap 都是簡單的鍵/值(key/value)對應表,但鍵只可以是 Object 型別,例如:

```
var wm1 = new WeakMap(),
    k1 = {},
    k2 = function () {},
    k3 = undefined;

wm1.set(k1,3);
wm1.get(k1); // 3
wm1.set(k2,4);
wm1.get(k2); // 4
wm1.get(k3); // undefined
```

Set

如果說 Map 類似 Object, 那麼也可以用 Array 去實作 Set, 但 Set 值不能重複亦不能直接提取某個位置的值,只可以知道有沒有這個值,如需要知道所有值則使用 forEach 迭代。

```
var s1 = new Set();
s1.add(1);
s1.add(5);
// Set { 1,5 }
```

但加入 Object 需要小心,可以看看以下例子

```
var s1 = new Set();
s1.add({c:3});
s1.add({c:3});
// Set { { c: 3 }, { c: 3 } }
```

這樣的話就會看似是重覆了,建議先賦值後加入,又或者使用 Map 去代替。

```
var s1 = new Set();
var o = { c: 3 };
s1.add(o);
s1.add(o);
// Set { { c: 3 } }
var s2 = new Set();
var m1 = new Map();
m1.set('c',3);
s2.add(m1);
// Set { Map { 'c' => 3 } }
```

WeakSet

WeakSet 的限制跟 WeakMap 一樣,只可以加入 Object 值而不能是原始數據 (只有 `{}` 以及 `function () {}`), 為何是 Weak, 因為 WeakSet 裡面所存儲的值都是被弱引用,所以如果沒有其他變量引用該值的話,就不能避免被回收掉 (garbage collection)。

```
var ws = new WeakSet();
ws.add({c:3});
ws.add(function(){});
```

Typed Arrays

向來 JavaScript 處理 Binary Data 都比較麻煩, Typed arrays 的出現就能夠使代碼快速處理這些數據。詳細代碼可以參照這裡

Generator

Generator 是一種函數,而這一種函數可以中途離開,下一次進入的時候則會載入上一次離開時的狀態(變量)。跟函數不同的是當調用 Generator 的時候,是返回一個 iterator, 當執行這個 `iterator.next()` 的時候才會執行 Generator 所定義的函數直至第一個 `yield` , `yield` 定義了所 return 的值。以下是一個訂單編號產生器:

```
function * orderIndexGenerator () {
  var index = 1;
  var startDay = new Date().toISOString().substring(0, 10);
  while (true) {
    let today = new Date().toISOString().substring(0, 10);
    // 如果下一次呼叫 .next() 時候已經過了一天的話,就需要更新預設值,那就確保
    if (startDay !== today) {
      startDay = today;
      index = 1;
    }
    yield startDay + '-' + index++;
  }
}
var oig = new oderIndexGenerator();
console.log(oig.next().value); // 2015-09-28-1
console.log(oig.next().value); // 2015-09-28-2
console.log(oig.next().value); // 2015-09-28-3
// 下一天再執行
console.log(oig.next().value); // 2015-09-29-1
```

Binary and Octal grammar

創建二進制數字的語法需要加上一個 leading zero, (0b 或 0B)。如果 0b 或 0B 後面的不是 0 或 1, 編譯時就會出現 `SyntaxError` 的錯誤。

```
var binaryNum = 0b3; // SyntaxError: Unexpected token ILLEGAL
```

同樣地創建八進制數字的語法需要加上一個 leading zero, (0o 或 0O)。如果 0o 或 0O 後面的不是 0,1,2,3,4,5,6,7, 編譯時就會出現 `SyntaxError` 的錯誤。

```
var octNum = 0b8; // SyntaxError: Unexpected token ILLEGAL
```

Extension for Object Literal

有經驗的開發者應該不難發現 ES6 的 Object 與先前提到的 class 十分相似,可以看看以下的代碼:

```
var protoObject = { key: 'value' };
var obj = {
    __proto__: protoObject,
    findSuperKey () {
        console.log(super.key); // 這裡的 super 就是指 __proto__
    }
};
```

換轉如果用 class 寫的話

```
'use strict';
class protoObject {
  constructor () {
    this.key = 'value';
  }
}

class obj extends protoObject {
  constructor () {
    super();
  }
  findSuperKey () {
    console.log(this.key);
  }
}

var o = new obj();
o.findSuperKey; // 'value'
```

另外 ES6 提供了一個快捷的方法去創建 Object, 就是如果當 Object key 的名稱跟變數的名稱是一樣的話,就可以縮短 Object 的代碼長度,減少冗餘。

```
// ES6 的語法糖
var a = 'apple', b = 'boy', c = 'cat';
var childrenVocab = {a, b, c};
// 以前的寫法
var a = 'apple', b = 'boy', c = 'cat';
var childrenVocab = { a: a, b: b, c: c };
```

Object 的鍵名也可以動態加入,不一定用 static string 來表示, 使代碼更容易擴展

```
var obj = {
  [(function(){return 'dymKey'})()] : 'dymKeyValue'
};
// { dymKey: 'dymKeyValue' }
```

New String methods

```
String.prototype.codePointAt
String.prototype.normalize
String.prototype.repeat
String.prototype.startsWith
String.prototype.endsWith
String.prototype.includes
String.prototype[Symbol.iterator]
// static methods
String.raw
String.fromCodePoint
```

Symbols

Symbol 是 ES6 所定義的第七種 JavaScript 基本類型,是一種不可變的數據型別,是對原始數據的封裝。

```
// 1. 基本應用,封裝原始數據,支援 typeof
var s = Symbol();
var s = Symbol('foo');
var s = Symbol(12);
var s = Symbol({ a: 1 });
typeof Symbol(); // 'symbol'

// 2. 使用 new 語法會拋出 TypeError 錯誤
var s = new Symbol(); // TypeError

// 3. 不能轉換成 string, number 或使用 JSON.stringify
var s = Symbol('foo');
s + 0; // TypeError: Cannot convert a Symbol value to a number
s + 'foo'; // TypeError: Cannot convert a Symbol value to a string

// 4. 每次創建都是新的
Symbol('foo') === Symbol('foo'); // false

// 5. 能夠封裝成 String
String(Symbol('foo')); // 'Symbol(foo)'

// 6. 可以用作 Object 的 key
var obj = {};
var sym = Symbol();
obj[sym] = 1;
console.log(obj[sym]); // 1
// 為了避免與 string key 有衝突, .keys 以及 .getOwnPropertyNames 均不會
Object.getOwnPropertyNames(obj); // []
Object.keys(obj); // []
Object.getOwnPropertySymbols(obj); // [ Symbol() ]

// 7. 註冊表
var symbol = Symbol.for('foo');
Symbol.for('foo') === symbol && Symbol.keyFor(symbol) === 'foo'; //
```

實際應用場景可以看看這裡

Template strings

簡單而言,這是一種語法糖,定義了多行字串(multi-lined string)的寫法,加入了以及加入標籤。

```
// Before ES6
var ms = 'A new line is then inserted.\nI am in the new line!';
// ES6 syntax sugar
var ms = `A new line is then inserted.
I am in the new line!`

// 模板字符串
var a = 1;
var b = 1;
// Before ES6
console.log(a + ' + ' + b + ' equals to ' + (a+b));
// ES6
console.log(`${a} + ${b} equals to ${a+b}`);
```

不過這裡會衍生安全性問題,由於 `${...}` 的寫法可以訪問變量內容,所以不能夠直接用作處理用戶端的輸入。

Arrow Functions

這並不是一種新的概念,這種匿名函數其實一直都在使用。

```
// Before ES6
var helloTargets = ['Alice','Bob','Cindy'];
helloTargets.map(function(target){
  console.log('Hello ' + target);
});
// Hello Alice
// Hello Bob
// Hello Cindy

// ES6 的代碼變得更簡潔
var helloTargets = ['Alice', 'Bob', 'Cindy'];
helloTargets.map((target) => console.log('Hello ' + target));

// 第二個例子
var psyTest = age => doingTest(age);
var psyTest = (age) => doingTest(age);
// 用 age => 或 (age) => 都是一樣效果
var psyTest = age,job => doingTest(age,job); // SyntaxError: Unexpe
var psyTest = (age,job) => doingTest(age,job);
// 如果沒有括號是有問題的，建議使用 (age) => 是為了方便日後代碼擴展
```

另外， `=>` 跟 `function` 有以下的不同：

- 傳統的 `function` 可以利用 `new` 來構造， `=>` 生成的函數就不能用 `new` 來構造。
- Lexical `this` , `super` , `arguments` , `new.target`
- `.bind` 對於 `=>` 是無效的

由此可見，它省卻了寫 `this` 的麻煩與迷思。

```
// Before ES6
function mother(){
  this.isAngry = true;
  this.callSonToDoHouseWork(function (){
    if(this.isAngry){ // undefined
      this.shopping();
    }
  });
}

// 需要定義 self 去解決這個問題
function mother(){
  this.isAngry = true;
  var self = this;
  this.callSonToDoHouseWork(() => {
    if(self.isAngry){ // true
      self.shopping();
    }
  });
}

// => with lexical this, 不需要定義 self
function mother(){
  this.isAngry = true;
  this.callSonToDoHouseWork(() => {
    if(this.isAngry){ // true
      this.shopping();
    }
  });
}
```

Promises

相信有寫過異步代碼 (Asynchronous) 的開發者對 Promise 應該不會陌生。它對於簡化代碼，解決 Callback hell， try/catch 無法抓到回調異常 (callback exception) 的問題的效果十分顯著。在先前的 Node.js 版本 (0.12) 已經有原生支持，當然還可以透過基於 Prmoises/A+ 標準所開發的第三方框架去實作起來，（例如 Q，bluebird 等)。

ES6 所定義的 Promise 有 4 種狀態， 分別是 Pending(待定)， Fulfilled(成功完成)， Rejected(失敗)， Settled(已經完成/失敗)。

```
// 基本語法
new Promise(function(resolve, reject) { ... });

// 例子
var p1 = new Promise(function(resolve, reject){
  resolve('finished'); // resolve 就是 fullfil promise !
});
var p2 = new Promise(function(resolve, reject){
  reject('exception p2'); // reject 就是 reject promise !
});

p1
.then(function(val){
  // .then 定義當 promise 被 fulfil 時應做什麼
  // 這個時候的狀態就是 settled
  console.log(val); // 'finished'
});

p2
.then(function(val){
  // 這個時候的狀態就是 settled
  console.log(val);
})
.catch(function(result){
  // 這個時候的狀態就是 settled
  // .catch 定義當 promise 被 reject 時應做什麼
  console.log(result); // 'exception p2'
});
```

除了 `.then` ， `.catch` 外，還有 `.all` 以及 `.race` 的方法，這個文檔暫時只提供基本 Promise 的應用而已。

for...of loops

這是一個語法糖，類似 C# 裡面的 `foreach(var item in items)` 。

```
let i1 = [1,2,3];
for(let i of i1){
  console.log(i);
}
/*
1
2
3
*/
let i2 = 'abc';
for(let i of i2){
  console.log(i);
}
/*
a
b
c
*/
```

此外，for...of 迴圈還支援下列樣式

```
// Generator instance
for(let i of (function*(){ yield 1; yield 2; yield 3; }())) { ... ]
// Generic iterable
for(let i of global.__createIterableObject([1, 2, 3])) { ... }
// Generic iterable instance
for(let i of Object.create(global.__createIterableObject([1, 2, 3])
```

結語

由於 ES6 剛發佈不久，不論前端或後端還是需要一定時間去調試和整合，大家可以去比較不同平台和瀏覽器目前的兼容性，這個文檔也會不停的更新。

參考資料

- ECMAScript compatibility table

- ES6 in Node.js

Node.js 基礎

前篇文章已經由介紹、安裝至設定都有完整介紹，Node.js 內部除了JavaScript 常用的函式(function)、物件(object)之外，也有許多不同的自訂物件，Node.js 預設建立這些物件為核心物件，是為了要讓開發流程更為，這些資料在官方文件已經具有許多具體說明。接下來將會介紹在開發Node.js 程式時常見的物件特性與使用方法。

Node.js http 伺服器建立

在 `Node.js官方網站 <http://Node.js.org>` 裡面有舉一個最簡單的HTTP 伺服器建立，一開始初步就是建立一個伺服器平台，讓 Node.js 可以與瀏覽器互相行為。每種語言一開始的程式建立都是以 Hello world 開始，最初也從 Hello world 帶各位進入 Node.js 的世界。

輸入以下程式碼，儲存檔案為 node_basic_http_hello_world.js

```
var server,
    ip   = "127.0.0.1",
    port = 1337,
    http = require('http');

server = http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
});

server.listen(port, ip);

console.log("Server running at http://" + ip + ":" + port);
```

程式碼解講，一開始需要有幾個基本的變數。

- ip: 機器本身的ip 位置，因為使用本地端，因此設定為 127.0.0.1
- port: 需要開通的阜號，通常設定為 http port 80，因範例不希望與基本 port 相

衝，隨意設定為 1337

在 Node.js 的程式中，有許多預設的模組可以使用，因此需要使用 require 方法將模組引入，在這邊我們需要使用 http 這個模組，因此將 http 載入。http 模組裡面內建有許多方法可以使用，這邊採用 createServer 創建一個基本的 http 伺服器，再將 http 伺服器給予一個 server 變數。裡面的回呼函式 (call back function) 可以載入http 伺服器的資料與回應方法 (request, response)。在程式裡面就可以看到我們直接回應給瀏覽器端所需的 Header，回應內容。

```
res.writeHead(200, {'Content-Type': 'text/plain'});
res.end('Hello World\n');
```

http 伺服器需要設定 port, ip，在最後需要設定 http 監聽，需要使用到 listen 事件，監聽所有 http 伺服器行為。

```
http.listen(port, ip);
```

所有事情都完成之後，需要確認伺服器正確執行因此使用 console ，在 JavaScript 裡就有這個原生物件， console 所印出的資料都會顯示於 Node.js 伺服器頁面，這邊印出的資料並不會傳送到使用者頁面上，之後許多除壞 (debug) 都會用到 console 物件。

```
console.log("Server running at http://" + ip + ":" + port);
```

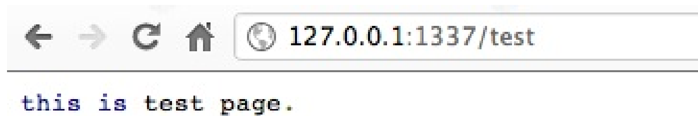
Node.js http 路徑建立

前面已經介紹如何建立一個簡單的 http 伺服器，接下來本章節將會介紹如何處理伺服器路徑 (route) 問題。 在 http 的協定下所有從瀏覽器發出的要求 (request) 都需要經過處理，路徑上的建立也是如此。

路徑就是指伺服器 ip 位置，或者是網域名稱之後，對於伺服器給予的要求。修改剛才的 hello world 檔案，修改如下。

```
server = http.createServer(function (req, res) {
  console.log(req.url);
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('hello world\n');
});
```

重新啟動 Node.js 程式後，在瀏覽器端測試一下路徑行為，結果如下圖：



當在瀏覽器輸入 http://127.0.0.1:1337/test ，在伺服器端會收到兩個要求，一個是我們輸入的 /test 要求，另外一個則是 /favicon.ico。 /test 的路徑要求， http 伺服器本身需要經過程式設定才有辦法回應給瀏覽器端所需要的回應，在伺服器中所有的路徑要求都是需要被解析才有辦法取得資料。 從上面解說可以了解到在 Node.js 當中所有的路徑都需要經過設定，未經過設定的路由會讓瀏覽器無法取得任何資料導致錯誤頁面的發生，底下將會解說如何設定路由，同時避免發生錯誤情形。 先前 Node.js 程式需要增加一些修改，才能讓使用者透過瀏覽器，在不同路徑時有不同的結果。根據剛才的程式做如下的修改，

```
var server,
ip   = "127.0.0.1",
port = 1337,
http = require('http'),
url = require('url'),
path;

server = http.createServer(function (req, res) {
  path = url.parse(req.url);

  res.writeHead(200, {'Content-Type': 'text/plain'});

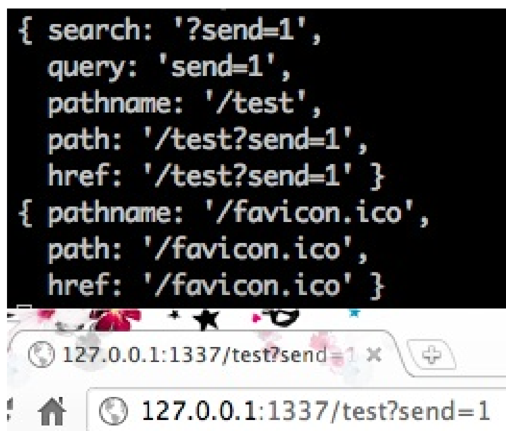
  switch (path.pathname) {
    case "/index":
      res.end('I am index.\n');
      break;
    case "/test":
      res.end('this is test page.\n');
      break;
    default:
      res.end('default page.\n');
      break;
  }
});

server.listen(port, ip);

console.log("Server running at http://" + ip + ":" + port);
```

程式做了片段的修改，首先載入 url 模組，另外增加一個 path 變數。 url 模組就跟如同他的命名一般，專門處理url 字串處理，裡面提供了許多方法來解決路徑上的問題。 因為從瀏覽器發出的要求路徑可能會帶有多種需求，或者GET 參數組合等。因此我們需要將路徑單純化，取用路徑部分的資料即可，例如使用者可能會送出 http://127.0.0.1:1337/test?send=1 ， 如果直接信任 **req.url** 就會收到結果為 /test?send=1 ， 所以需要透過 url 模組的方法將路徑資料過濾。

在這邊使用 url.parse 的方法，裡面帶入網址格式資料，會回傳路徑資料。為了後需方便使用，將回傳的資料設定到 path 變數當中。在回傳的路徑資料，裡面包含資訊，如下圖，



這邊只需要使用單純的路徑要求，直接取用 path.pathname ，就可以達到我們的目的。

最後要做路徑的判別，在不同的路徑可以指定不同的輸出，在範例中有三個可能結果，第一個從瀏覽器輸入 /index 就會顯示 index 結果， /test 就會呈現出 test 頁面，最後如果都不符合預期的輸入會直接顯示 default 的頁面，最後的預防可以讓瀏覽器不會出現非預期結果，讓程式的可靠性提昇，底下為測試結果。



Node.js 檔案讀取

前面已經介紹如何使用路由（rount）做出不同的回應，實際應用只有在瀏覽器只有輸出幾個文字資料總是不夠的，在本章節中將介紹如何使用檔案讀取，輸出檔案資料，讓使用者在前端瀏覽器也可以讀取到完整的 HTML, CSS, JavaScript 檔案輸出。

檔案管理最重要的部分就是 `File system <http://node.js.org/docs/latest/api/fs.html>` 這個模組，此模組可以針對檔案做管理、監控、讀取等行為，裡面有許多預設的方法，底下是檔案輸出的基本範例，底下會有兩個檔案，第一個是靜態 HTML 檔案，

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh-tw" lang="z
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF
    <title>Node.js index html file</title>
</head>
<body>
    <h1>Node.js index html file</h1>
</body>
</html>
```

另一個為 Node.js 程式，

```
var fs = require("fs"),
filename = "static/index.html",
encode = "utf8";

fs.readFile(filename, encode, function(err, file) {
  console.log(file);
});
```

一開始直接載入**file system** 模組，載入名稱為 **fs** 。讀取檔案主要使用的方法為 readFile ，裡面以三個參數 路徑**(file path)** , 編碼方式**(encoding)** ，回應函式 **(callback)** ，路徑必須要設定為靜態html 所在位置，才能指定到正確的檔案。靜態檔案的編碼方式也必須正確，這邊使用靜態檔案的編碼為 **utf8** ，如果編碼設定錯誤，Node.js 讀取出來檔案結果會使用 byte raw 格式輸出，如果 錯誤編碼格式，會導致輸出資料為 **byte raw**

```
1. clonn@clonn-Ma
clonn@clonn-MacBook-Air:~/node/nodejs-book/src$ node node_basic_file_simple.js
<Buffer 3c 21 44 4f 43 54 59 50 45 20 68 74 6d 6c 3e 0a 3c 68 74 6d 6c 20 78 6d
```

回應函式 中裡面會使用兩個變數，error 為錯誤資訊，如果讀取的檔案不存在，或者發生錯誤，error 數值會是 true ，如果成功讀取資料 error 將會是 false 。content 則是檔案內容，資料讀取後將會把資料全數丟到content 這個變數當中。

最後程式的輸出結果畫面如下，

```
clonn@clonn-MacBook-Air:~/node/nodejs-book/src$ node node_basic_file_simple.js
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh-CN" lang="zh-CN">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-7" />
    <title>node.js index static file</title>
</head>
<body>
    <h1>node.js index static file</h1>
</body>
</html>
```

Node.js http 靜態檔案輸出

前面已經了解如何讀取本地端檔案，接下來將配合http 伺服器路由，讓每個路由都能夠輸出相對應的靜態 html 檔案。 首先新增加幾個靜態html 檔案：

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh-TW" lang="z
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF
    <title>Node.js index html file</title>
</head>
<body>
    <h1>Node.js index html file</h1>
</body>
</html>
```

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh-TW" lang="z
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF
    <title>Node.js test html file</title>
</head>
<body>
    <h1>Node.js test html file</h1>
</body>
</html>
```

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh-TW" lang="z
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF
    <title>Node.js static html file</title>
</head>
<body>
    <h1>Node.js static html file</h1>
</body>
</html>
```

準備一個包含基本路由功能的http 伺服器

```
var server,
    ip   = "127.0.0.1",
    port = 1337,
    http = require('http'),
    url = require('url');

server = http.createServer(function (req, res) {
  var path = url.parse(req.url);
});

server.listen(port, ip);

console.log("Server running at http://" + ip + ":" + port);
```

加入 **file system** 模組， 使用 **readFile** 的功能，將這一段程式放置於createServer 的回應函式中。

```
fs.readFile(filePath, encode, function(err, file) {
});
```

readFile 的回應函式裡面加入頁面輸出，讓瀏覽器可以正確讀到檔案，在這邊我們設定讀取的檔案為 html 靜態檔案，所以 Content-type 設定為 **text/html** 。讀取到檔案的內容，將會正確輸出成 html 靜態檔案。

```
fs.readFile(filePath, encode, function(err, file) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write(file);
  res.end();
});
```

到這邊為止基本的程式內容都已經完成，剩下一些細節的調整。首先路徑上必須做調整，目前的靜態檔案全部都放置於 **static** 資料夾 底下，設定一個變數來記住資料夾位置。

接著將瀏覽器發出要求路徑與資料夾組合，讀取正確html 靜態檔案。使用者有可能會輸入錯誤路徑，所以在讀取檔案的時候要加入錯誤處理，同時回應 **404** 伺服器無法正確回應的 http header 格式。

加入這些細節的修改，一個基本的http 靜態 html 輸出伺服器就完成了，完整程式碼如下，

```
var server,
    ip   = "127.0.0.1",
    port = 1337,
    http = require('http'),
    fs = require("fs"),
    folderPath = "static",
    url = require('url'),
    path,
    filePath,
    encode = "utf8";

server = http.createServer(function (req, res) {
  path = url.parse(req.url);
  filePath = folderPath + path.pathname;

  fs.readFile(filePath, encode, function(err, file) {
    if (err) {
        res.writeHead(404, {'Content-Type': 'text/plain'});
        res.end();
        return;
    }

    res.writeHead(200, {'Content-Type': 'text/application'});
    res.write(file);
    res.end();
  });
});

server.listen(port, ip);

console.log("Server running at http://" + ip + ":" + port);
```

Node.js http GET 資料擷取

http 伺服器中，除了路由之外另一個最常使用的方法就是擷取GET 資料。本單元將會介紹如何透過基本 http 伺服器擷取瀏覽器傳來的要求，擷取 GET 資料。

在 http 協定中，GET 參數都是藉由 URL 從瀏覽器發出要求送至伺服器端，基本的傳送網址格式可能如下，

```
http://127.0.0.1/test?send=1&test=2
```

上面這段網址，裡面的 GET 參數就是 send 而這個變數的數值就為 1，如果想要在 http 伺服器取得GET 資料，需要在瀏覽器給予的要求 (request) 做處理，

首先需要載入 **query string** 這個模組，這個模組主要是用來將字串資料過濾後，轉換成 **JavaScript** 物件。

```
qs = require('querystring');
```

接著在第一階段，利用 url 模組過濾瀏覽器發出的 URL 資料後，將回應的物件裡面的 query 這個變數，是一個字串值，資料過濾後如下，

```
send=1&test=2
```

透過 query string ，使用 parse 這個方法將資料轉換成JavaScript 物件，就表示 GET 的資料已經被伺服器端正式擷取下來，

```
path = url.parse(req.url);
parameter = qs.parse(path.query);
```

整個 Node.js http GET 參數完整擷取程式碼如下，

```
var server,
    ip   = "127.0.0.1",
    port = 1337,
    http = require('http'),
    qs = require('querystring'),
    url = require('url');

server = http.createServer(function (req, res) {
  var path = url.parse(req.url),
      parameter = qs.parse(path.query);

  console.dir(parameter);

  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.write('Browser test GET parameter\n');
  res.end();
});

server.listen(port, ip);

console.log("Server running at http://" + ip + ":" + port);
```

程式運作之後，由瀏覽器輸入要求網址之後，伺服器端回應資料為，

```
Server running at http://127.0.0.1:1337
{ send: '1', test: '1' }
```

本章結語

前面所解說的部份，一大部分主要是處理 http 伺服器基本問題，雖然在某些部分有牽扯到 http 伺服器基本運作原理，主要還是希望可以藉由這些基本範例練習 Node.js ，練習回應函式與語法串接的特點，習慣編寫 JavaScript 風格程式。當然直接這樣開發 Node.js 是非常辛苦的，接下來在模組實戰開發的部份將會介紹特定的模組，一步一步帶領各位從無到有進行 Node.js 應用程式開發。

NPM 套件管理工具

npm 全名為 **N** ode **P** ackage **M** anager， 是 Node.js 的套件（package）管理工具， 類似 Perl 的 ppm 或 PHP 的 PEAR 等。 安裝 npm 後， 使用 `npm install module_name` 指令即可安裝新套件， 維護管理套件的工作會更加輕鬆。

npm 可以讓 Node.js 的開發者， 直接利用、擴充線上的套件庫（packages registry）， 加速軟體專案的開發。 npm 提供很友善的搜尋功能， 可以快速找到、安裝需要的套件， 當這些套件發行新版本時， npm 也可以協助開發者自動更新這些套件。

npm 不僅可用於安裝新的套件，它也支援搜尋、列出已安裝模組及更新的功能。

安裝 NPM

nvm在安裝Node.js時即會內建npm，請先參考NODE_INSTALL

NPM 安裝後測試

npm 是指令列工具（command-line tool）， 使用時請先打開系統的文字終端機工具。

測試 npm 安裝與設定是否正確，請輸入指令如下：

```
npm -v
```

或是：

```
npm --version
```

如果 npm 已經正確安裝設定，就會顯示版本訊息：

- 執行結果（範例）

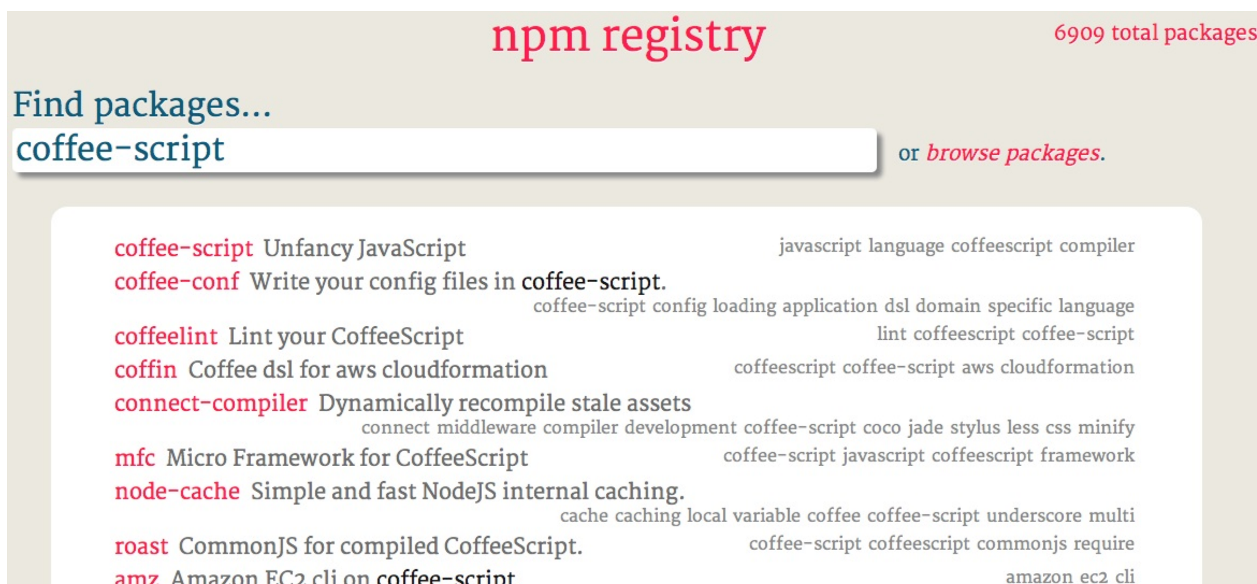
```
1.1.0-2
```

使用 NPM 安裝套件

npm 目前擁有超過 6000 種套件（packages）， 可以在 `npm registry` <https://www.npmjs.com/> 使用關鍵字搜尋套件。

https://www.npmjs.com/

舉例來說，在關鍵字欄位輸入「coffee-script」， 下方的清單就會自動列出包含 coffee-script 關鍵字的套件。



接著我們回到終端機模式的操作， `npm` 的指令工具本身就可以完成套件搜尋的任務。

例如，以下的指令同樣可以找出 coffee-script 相關套件。

```
npm search coffee-script
```

以下是搜尋結果的參考畫面：

```
npm http GET https://registry.npmjs.org/-/all/since?stale=update_after&startkey=1328322837000
npm http 200 https://registry.npmjs.org/-/all/since?stale=update_after&startkey=1328322837000
NAME                 DESCRIPTION                                              AUTHOR        DATE              KEYWORDS
amz                  Amazon EC2 cli on coffee-script                          =selead       2012-01-31 16:39  amazon ec2 cli
coffee-conf          Write your config files in coffee-script.               =MSNexploder  2011-11-20 21:43  coffee-script c
coffee-script        Unfancy JavaScript                                       =jashkenas    2012-01-13 23:07  javascript lang
coffee-toaster       Minimalist dependency management system for coffee-script.  =nybras    2011-10-29 12:25
coffeeapp            CoffeeApp wrapper (handling coffee-script) for CouchApp (http://couchapp.org/). =andrzejsliwa 2011-01-28 23:16
coffeelint           Lint your CoffeeScript                                   =clutchski    2012-01-26 05:56  lint coffeescri
coffeescript-notify  Coffee-Script notify tool for coffee based on the coffeescript-growl tool =bdryanovski 2011-09-17 14:34  coffe
coffin               Coffee dsl for aws cloudformation                        =chrisfjones  2012-01-04 20:12  coffeescript co
connect-compiler     Dynamically recompile stale assets                       =dsc          2011-12-26 15:43  connect middlew
cupcake              Quick Coffee-Script Web App Template Generator           =jackhq       2012-02-04 01:52  javascript expr
gesundheit           Concise SQL generation in coffee-script                  =grncdr       2012-02-01 06:07
iced-coffee-script   IcedCoffeeScript                                         =maxtaco      2012-02-04 01:09  javascript lang
mfc                  Micro Framework for CoffeeScript                         =tadeuzagallo 2012-01-17 17:34  coffee-script j
node-cache           Simple and fast NodeJS internal caching.                 =tcs-de       2011-10-20 14:52  cache caching l
roast                CommonJS for compiled CoffeeScript.                      =chrislloyd   (prehistoric)     coffee-script c
stitcher             coffee-script js less css eco commonJs stitcher.         =sunliutao    2012-01-17 09:02
tamed-coffee-script  Unfancy JavaScript                                       =maxtaco      2011-12-12 22:01  javascript lang
tiers                Web framework built on coffee-script.                    =terryvesper  2011-03-14 22:38  tiers web frame
```

找到需要的套件後（例如 express），即可使用以下指令安裝：

```
npm install coffee-script
```

值得注意的一點是，使用 `npm install` 會將指定的套件， 安裝在工作目錄（Working Directory）的 `node_modules` 資料夾下。

以 Windows 為例，如果執行 `npm install` 的目錄位於：

`C:\project1`

那麼 npm 將會自動建立一個 node_modules 的子目錄（如果不存在）。

`C:\project1\node_modules`

並且將下載的套件，放置於這個子目錄，例如：

`C:\project1\node_modules\coffee-script`

這個設計讓專案可以個別管理相依的套件， 並且可以在專案佈署或發行時， 將這些套件（位於 node_modules）一併打包， 方便其它專案的使用者不必再重新下載套件。

這個 `npm install` 的預設安裝模式為 **local** (本地)， 只會變更當前專案的資料夾， 不會影響系統。

另一種安裝模式稱為 **global** （全域）， 這種模式會將套件安裝到系統資料夾， 也就是 npm 安裝路徑的 `node_modules` 資料夾， 例如：

`C:Program Filesnodejsnode_modules`

是否要使用全域安裝， 可以依照套件是否提供 新指令 來判斷， 舉例來說， express 套件提供 `express` 這個指令， 而 coffee-script 則提供 `coffee` 指令。

在 local 安裝模式中，這些指令的程式檔案， 會被安裝到 `node_modules` 的 `.bin` 這個隱藏資料夾下。 除非將 .bin 的路徑加入 PATH 環境變數， 否則要執行這些指令將會相當不便。

為了方便指令的執行， 我們可以在 `npm install` 加上 `-g` 或 `--global` 參數， 啟用 global 安裝模式。例如：

```
npm install -g coffee-script
npm install -g express
```

使用 global 安裝模式， 需要注意執行權限與搜尋路徑的問題， 若權限不足，可能會出現類似以下的錯誤訊息：

```
npm ERR! Error: EACCES, permission denied '...'
npm ERR!
npm ERR! Please try running this command again as root/Administ
```

要獲得足夠得執行權限，請參考以下說明：

- Windows 7 或 2008 以上，在「命令提示字元」的捷徑按右鍵， 選擇「以系統管理員身分執行」， 執行 npm 指令時就會具有 Administrator 身分。
- Mac OS X 或 Linux 系統，可以使用 `sudo` 指令，例如：

  `sudo npm install -g express`

- Linux 系統可以使用 root 權限登入，或是以「 `sudo su -` 」切換成 root 身分。 （使用 root 權限操作系統相當危險，因此並不建議使用這種方式。）

若加上 `-g` 參數，使用 `npm install -g coffee-script` 完成安裝後， 就可以在終端機執行 `coffee` 指令。例如：

```
coffee -v
```

- 執行結果（範例）

```
CoffeeScript version 1.2.0
```

若未將 Node.js 套件安裝路徑加入環境變數 NODE_PATH，在引入時會回報錯誤。

- 報錯範例

```
module.js:340
    throw err;
          ^
Error: Cannot find module 'express'
    at Function.Module._resolveFilename (module.js:338:15)
    at Function.Module._load (module.js:280:25)
    at Module.require (module.js:362:17)
    at require (module.js:378:17)
    at Object.<anonymous> (/home/clifflu/test/node.js/httpd/express
    at Module._compile (module.js:449:26)
    at Object.Module._extensions..js (module.js:467:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Module.runMain (module.js:492:10)
```

- 使用 ubuntu PPA 安裝 Node.js 的設定範例

```
echo 'NODE_PATH="/usr/lib/node_modules"' | sudo tee -a /etc/enviror
```

套件的更新及維護

除了前一節說明的 search 及 install 用法， npm 還提供其他許多指令（commands）。

使用 `npm help` 可以查詢可用的指令。

```
npm help
```

- 執行結果（部分）

```
where <command> is one of:
    adduser, apihelp, author, bin, bugs, c, cache, completion,
    config, deprecate, docs, edit, explore, faq, find, get,
    help, help-search, home, i, info, init, install, la, link,
    list, ll, ln, login, ls, outdated, owner, pack, prefix,
    prune, publish, r, rb, rebuild, remove, restart, rm, root,
    run-script, s, se, search, set, show, star, start, stop,
    submodule, tag, test, un, uninstall, unlink, unpublish,
    unstar, up, update, version, view, whoami
```

使用 `npm help command` 可以查詢指令的詳細用法。例如：

```
npm help list
```

接下來，本節要介紹開發過程常用的 npm 指令。

使用 `list` 可以列出已安裝套件：

```
npm list
```

- 執行結果（範例）

```
├── coffee-script@1.2.0
└─┬ express@2.5.6
  ├─┬ connect@1.8.5
  │ └── formidable@1.0.8
  ├── mime@1.2.4
  ├── mkdirp@0.0.7
  └── qs@0.4.1
```

檢視某個套件的詳細資訊，例如：

```
npm show express
```

升級所有套件（如果該套件已發佈更新版本）：

```
npm update
```

升級指定的套件：

```
npm update express
```

移除指定的套件：

```
npm uninstall express
```

使用 package.json

對於正式的 Node.js 專案， 可以建立一個命名為 `package.json` 的設定檔（純文字格式）， 檔案內容參考範例如下：

package.json（範例）

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "2.5.5",
    "coffee-script": "latest",
    "mongoose": ">= 2.5.3"
  }
}
```

亦可使用NPM內建的REPL(Read-Eval-Print Loop)產生package.json檔

```
    npm init
```

其中 `name` 與 `version` 依照專案的需求設置。

需要注意的是 `dependencies` 的設定， 它用於指定專案相依的套件名稱及版本：

- `"express": "2.5.5"`

  //代表此專案相依版本 2.5.5 的 express 套件

- `"coffee-script": "latest"`

  //使用最新版的 coffee-script 套件（每次更新都會檢查新版）

- `"mongoose": ">= 2.5.3"`

  //使用版本大於 2.5.3 的 mongoose 套件

假設某個套件的新版可能造成專案無法正常運作， 就必須指定套件的版本， 避免專案的程式碼來不及更新以相容新版套件。 通常在開發初期的專案， 需要盡可能維持新套件的相容性（以取得套件的更新或修正）， 可以用「 `>=` 」設定最低相容的版本， 或是使用「 `latest` 」設定永遠保持最新套件。

Express 介紹

在前面的Node.js 基礎當中介紹許多許多開設http 的使用方法及介紹，以及許多基本的Node.js 基本應用。

接下來要介紹一個套件稱為express Express ，這個套件主要幫忙解決許多Node.js http server 所需要的基本服務，讓開發http service 變得更為容易，不需要像之前需要透過層層模組（module）才有辦法開始編寫自己的程式。

這個套件是由TJ Holowaychuk 製作而成的套件，裡面包含基本的路由處理(route)，http 資料處理（GET/POST/PUT），另外還與樣板套件（js html template engine）搭配，同時也可以處理許多複雜化的問題。

Express 安裝

安裝方式十分簡單，只要透過之前介紹的 NPM 就可以使用簡單的指令安裝，指令如下，

```
npm install -g express
```

這邊建議需要將此套件安裝成為全域模組，方便日後使用。

Express 基本操作

express 的使用也十分簡單，先來建立一個基本的hello world ，

```
var app = require('express')();
var port = 1337;

app.listen(port);

app.get('/', function(req, res){
  res.send('hello world');
});

console.log('start express server\n');
```

可以從上面的程式碼發現，基本操作與Node.js http 的建立方式沒有太大差異，主要差在當我們設定路由時，可以直接透過 app.get 方式設定回應與接受方式。

Express 路由處理

Express 對於 http 服務上有許多包裝，讓開發者使用及設定上更為方便，例如有幾個路由設定，那我們就統一藉由 app.get 來處理，

```
// ... Create http server

app.get('/', function(req, res){
  res.send('hello world');
});

app.get('/test', function(req, res){
  res.send('test render');
});

app.get('/user/', function(req, res){
  res.send('user page');
});
```

如上面的程式碼所表示，app.get 可以帶入兩個參數，第一個是路徑名稱設定，第二個為回應函式(call back function)，回應函式裡面就如同之前的 createServer 方法，裡面包含 request， response 兩個物件可供使用。使用者就可以透過瀏覽器，

輸入不同的url 切換到不同的頁面，顯示不同的結果。

路由設定上也有基本的配對方式，讓使用者從瀏覽器輸入的網址可以是一個變數，只要符合型態就可以有對應的頁面產出，例如，

```
// ... Create http server

app.get('/user/:id', function(req, res){
  res.send('user: ' + req.params.id);
});

app.get('/:number', function(req, res){
  res.send('number: ' + req.params.number);
});
```

裡面使用到:number， 從網址輸入之後就可以直接使用 req.params.number 取得所輸入的資料，變成url 參數使用，當然前面也是可以加上路徑的設定， /user/:id，在瀏覽器上路徑必須符合 /user/xxx，透過 req.params.id 就可以取到 xxx這個字串值。

另外，express 參數處理也提供了路由參數配對處理，也可以透過正規表示法作為參數設定，

```
var app = require('express').createServer(),
    port = 1337;

app.listen(port);

app.get(/^\/ip?(?:\/(\d{2,3})(?:\.(\d{2,3}))(?:\.(\d{2,3}))(?:\.(\d
  res.send(req.params);
});
```

上面程式碼，可以發現後面路由設定的型態是正規表示法，裡面設定格式為 /ip 之後，必須要加上ip 型態才會符合資料格式，同時取得ip資料已經由正規表示法將資料做分群，因此可以取得ip的四個數字。

此程式執行之後，可以透過瀏覽器測試，輸入網址為 localhost:3000/ip/255.255.100.10，可以從頁面獲得資料，

```
    [
        "255",
        "255",
        "100",
        "10"
    ]
```

此章節全部範例程式碼如下，

```
/**
 * @overview
 *
 * @author Caesar Chi
 * @blog clonn.blogspot.com
 * @version 2012/02/26
 */

// create server.
var app = require('express').createServer(),
    port = 1337;

app.listen(port);

// normal style
app.get('/', function(req, res){
  res.send('hello world');
});

app.get('/test', function(req, res){
    res.send('test render');
});

// parameter style
app.get('/user/:id', function(req, res){
  res.send('user: ' + req.params.id);
});
```

```
app.get('/:number', function(req, res){
  res.send('number: ' + req.params.number);
});

// REGX style
app.get(/^\/ip?(?:\/(\d{2,3})(?:\.(\d{2,3}))(?:\.(\d{2,3})))?/, fur
  res.send(req.params);
});

app.get('*', function(req, res){
  res.send('Page not found!', 404);
});

console.log('start express server\n');
```

Express middleware

Express 裡面有一個十分好用的應用概念稱為middleware，可以透過 middleware 做出複雜的效果，同時上面也有介紹 next 方法參數傳遞，就是靠 middleware 的概念來傳遞參數，讓開發者可以明確的控制程式邏輯。

```
// .. create http server
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(express.session());
```

上面都是一種 middleware 的使用方式，透過 app.use 方式裡面載入函式執行方法，回應函式會包含三個基本參數，response， request， next，其中next 表示下一個 middleware 執行函式，同時會自動將預設三個參數繼續帶往下個函式執行，底下有個實驗，

上面的片段程式執行後，開啟瀏覽器，連結上 localhost:1337/，會發現伺服器回應結果順序如下，

```
first middle ware
second middle ware
execute middle ware
end middleware function
```

從上面的結果可以得知，剛才設定的 middleware 都生效了，在 app.use 設定的 middleware 是所有url 皆會執行方法，如果有指定特定方法，就可以使用 app.get 的 middleware 設定，在 app.get 函式的第二個參數，就可以帶入函式，或者是匿名函式，只要函式裡面最後會接受 request, response, next 這三個參數，同時也有正確指定 next 函式的執行時機，最後都會執行到最後一個方法，當然開發者也可以評估程式邏輯要執行到哪一個階段，讓邏輯可以更為分明。

Express 路由應用

在實際開發上可能會遇到需要使用參數等方式，混和變數一起使用，express 裡面提供了一個很棒的處理方法 app.all 這個方式，可以先採用基本路由配對，再將設定為每個不同的處理方式，開發者可以透過這個方式簡化自己的程式邏輯，

```
/**
 * @overview
 *
 * @author Caesar Chi
 * @blog clonn.blogspot.com
 * @version 2012/02/26
 */

// create server.
var app = require('express').createServer(),
    port = 1337;

app.listen(port);

// normal style
app.get('/', function(req, res){
  res.send('hello world');
});
```

```
app.get('/test', function(req, res){
  res.send('test render');
});

// parameter style
app.get('/user/:id', function(req, res){
  res.send('user: ' + req.params.id);
});

app.get('/:number', function(req, res){
  res.send('number: ' + req.params.number);
});

// REGX style
app.get(/^\/ip?(?:\/(\d{2,3})(?:\.(\d{2,3}))(?:\.(\d{2,3})))?/, fur
  res.send(req.params);
});

app.get('*', function(req, res){
    res.send('Page not found!', 404);
});

console.log('start express server\n');
```

內部宣告一組預設的使用者分別給予名稱設定，藉由app.all 這個方法，可以先將路由雛形建立，再接下來設定 app.get 的路徑格式，只要符合格式就會分配進入對應的方法中，像上面的程式當中，如果使用者輸入路徑為 /user/0 ，除了執行 app.all 程式之後，執行next 方法就會對應到路徑設定為 /user/:id 的這個方法當中。如果使用者輸入路徑為 /user/0/edit ，就會執行到 /user/:id/edit 的對應方法。

Express GET 應用範例

我們準備一個使用GET方法傳送資料的表單。

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset
        <title>Node.js菜鳥筆記(1)</title>
        <link rel="stylesheet" href="css/style.css" type="text/css'
    </head>
    <body>
        <h1>Node.js菜鳥筆記-註冊</h1>
        <form id="signup" method="GET" action="http://localhost:30(
            <label>使用者名稱：</label><input type="text" id="usernar
            <label>電子郵件：</label><input type="text" id="email" n
            <input type="submit" value="註冊我的帳號" /><br>
        </form>
    </body>
</html>
```

這個表單沒有什麼特別的地方，我們只需要看第9行，form使用的method是GET，然後action是"http://localhost:3000/Signup"，等一下我們要來撰寫/Signup這個URL Path的處理程式。

處理 *Signup* 行為

我們知道所謂的GET方法，會透過URL來把表單的值給帶過去，以上面的表單來說，到時候URL會以這樣的形式傳遞

```
http://localhost:3000/Signup?username=xxx&email=xxx
```

所以要能處理這樣的資料，必須有以下功能:

- 解析URL
- 辨別動作是Signup
- 解析出username和email

一旦能取得username和email的值，程式就能加以應用了。

處理 Signup 的程式碼雛形，

```
// load module
var url  = require('url');

urlData = url.parse(req.url,true);
action = urlData.pathname;
res.writeHead(200, {"Content-Type":"text/html; charset=utf-8"});

if (action === "/Signup") {
    user = urlData.query;
    res.end("<h1>" + user.username + "歡迎您的加入</h1><p>我們已經將會
}
```

首先需要加載 url module，它是用來協助我們解析URL的模組，接著使用 url.parse 方法，第一個傳入url 字串變數，也就是req.url。另外第二個參數的用意是，設為ture則引進 querystring模組來協助處理，預設是false。它影響到的是 urlData.query，設為true會傳回物件，不然就只是一般的字串。url.parse 會將字串內容整理成一個物件，我們把它指定給urlData。

action 變數作為記錄pathname，這是我們稍後要來判斷目前網頁的動作是什麼。接著先將 html 表頭資訊 (Header)準備好，再來判斷路徑邏輯，如果是 */Signup* 這個動作，就把urlData.query裡的資料指定給user，然後輸出user.username和 user.email，把使用者從表單註冊的資料顯示於頁面中。

最後進行程式測試，啟動 Node.js 主程式之後，開啟瀏覽器就會看到表單，填寫完畢按下送出，就可以看到結果了。

完整 Node.js 程式碼如下，

```
var http = require('http'),
    url  = require('url'),
    fs   = require("fs"),
    server;

server = http.createServer(function (req,res) {
  var urlData,
      encode   = "utf8",
      filePath = "view/express_get_example_form.html",
      action;

  urlData = url.parse(req.url,true);
  action = urlData.pathname;
  res.writeHead(200, {"Content-Type":"text/html; charset=utf-8"});

  if (action === "/Signup") {
        user = urlData.query;
        res.end("<h1>" + user.username + "歡迎您的加入</h1><p>我們已經
  }
  else {
        fs.readFile(filePath, encode, function(err, file) {

        res.write(file);
        res.end();
        });
    }
});

server.listen(3000);

console.log('Server跑起來了，現在時間是:' + new Date());
```

Express POST 應用範例

一開始準備基本的 html 表單，傳送內容以 POST 方式， form 的 action 屬性設定為 POST，其餘 html 內容與前一個範例應用相同，

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset
        <title>Node.js菜鳥筆記(1)</title>
        <link rel="stylesheet" href="css/style.css" type="text/css'
    </head>
    <body>
        <h1>Node.js菜鳥筆記-註冊</h1>
        <form id="signup" method="POST" action="http://localhost:3(
            <label>使用者名稱：</label><input type="text" id="usernar
            <label>電子郵件：</label><input type="text" id="email" n
            <input type="submit" value="註冊我的帳號" /><br>
        </form>
    </body>
</html>
```

Node.js 的程式處理邏輯與前面 GET 範例類似，部分程式碼如下，

```
qs   = require('querystring'),

if (action === "/Signup") {
  formData = '';
  req.on("data", function (data) {

    formData += data;

  });

  req.on("end", function () {
    user = qs.parse(formData);
    res.end("<h1>" + user.username + "歡迎您的加入</h1><p>我們已經將會
  });
}
```

主要加入了'querystring' 這個module，方便我們等一下解析由表單POST回來的資料，另外加入一個formData的變數，用來搜集待等一下表單回傳的資料。前面的GET 範例，我們只從req 拿出url的資料，這次要在利用 req 身上的事件處理。

JavaScript在訂閱事件時使用addEventListener，而Node.js使用的則是on。這邊加上了監聽 *data* 的事件，會在瀏覽器傳送資料到 Web Server時被執行，參數是它所接收到的資料，型態是字串。

接著再增加 *end* 的事件，當瀏覽器的請求事件結束時，它就會動作。

由於瀏覽器使用POST在上傳資料時，會將資料一塊塊地上傳，因為我們在監聽data事件時，透過formData 變數將它累加起來< 不過由於我們上傳的資料很少，一次就結束，不過如果日後需要傳的是資料比較大的檔案，這個累加動作就很重要。

當資料傳完，就進到end事件中，會用到 qs.parse來解析formData。formData的內容是字串，內容是：

```
username=wordsmith&email=wordsmith%40some.where
```

而qs.parse可以幫我們把這個querystring轉成物件的格式，也就是：

```
{username=wordsmith&email=wordsmith%40some.where}
```

一旦轉成物件並指定給user之後，其他的事情就和GET方法時操作的一樣，寫response的表頭，將內容回傳，並將user.username和user.email代入到內容中。

修改完成後，接著執行 Node.js 程式，啟動 web server ，開啟瀏覽器進入表單測試看看，POST 的方式能否順利運作。

完整程式碼如下，

```
var http = require('http'),
    url  = require('url'),
    fs   = require("fs"),
    qs   = require('querystring'),
    server;

server = http.createServer(function (req,res) {
  var urlData,
```

```
        encode   = "utf8",
        filePath = "view/express_post_example_form.html",
        formData,
        action;

    urlData = url.parse(req.url,true);
    action = urlData.pathname;
    res.writeHead(200, {"Content-Type":"text/html; charset=utf-8"});

    if (action === "/Signup") {
      formData = '';
      req.on("data", function (data) {

        formData += data;

      });

      req.on("end", function () {
        user = qs.parse(formData);
        res.end("<h1>" + user.username + "歡迎您的加入</h1><p>我們已經將
      });
    }
    else {
      fs.readFile(filePath, encode, function(err, file) {

        res.write(file);
        res.end();
      });
    }

  });

  server.listen(3000);

  console.log('Server跑起來了，現在時間是:' + new Date());
```

Express AJAX 應用範例

在Node.js要使用Ajax傳送資料，並且與之互動，在接受資料的部份沒有太大的差別，client端不是用GET就是用POST來傳資料，重點在處理完後，用JSON格式回傳。當然Ajax不見得只傳JSON格式，有時是回傳一段HTML碼，不過後者對伺服器來說，基本上就和前兩篇沒有差別了。所以我們還是以回傳JSON做為這一回的主題。

這一回其實大多數的工作都會落在前端Ajax上面，前端要負責發送與接收資料，並在接收資料後，撤掉原先發送資料的表單，並將取得的資料，改成HTML格式之後，放上頁面。

首先先準備 HTML 靜態頁面資料，

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset
        <title>Node.js菜鳥筆記(1)</title>
        <link rel="stylesheet" href="css/style.css" type="text/css'
    </head>
    <body>
        <h1>Node.js菜鳥筆記-註冊</h1>
        <form id="signup" method="POST" action="http://localhost:3(
            <label>使用者名稱：</label><input type="text" id="usernar
            <label>電子郵件：</label><input type="text" id="email" n
            <input type="submit" value="註冊我的帳號" /><br>
        </form>
    </body>
</html>
```

HTML 頁面上準備了一個表單，用來傳送註冊資料。接著直接引用了 Google CDN 來載入 jQuery，用來幫我們處理 Ajax 的工作，這次要傳送和接收的工作，很大的變動都在 HTML 頁面上的 JavaScript當中。我們要做的事有(相關 jQuery 處理這邊不多做贅述，指提起主要功能解說)：

- 用jQUery取得submit按鈕，綁定它的click動作
- 取得表單username和email的值，存放在user這個物件中
- 用jQuery的$.post方法，將user的資料傳到Server
- 一旦成功取得資料後，透過greet這個function，組成回報給user的訊息

- 清空原本給使用者填資料的表單
- 將Server回傳的username、email和id這3個資料，組成回應的訊息
- 將訊息放到原本表單的位置

經過以上的處理後，一個Ajax的表單的基本功能已經完成。

接著進行 Node.js 主要程式的編輯，部分程式碼如下，

```
var fs   = require("fs"),
    qs   = require('querystring');

if (action === "/Signup") {
  formData = '';
  req.on("data", function (data) {

    formData += data;

  });

  req.on("end", function () {
    var msg;

    user = qs.parse(formData);
    user.id = "123456";
    msg = JSON.stringify(user);
    res.writeHead(200, {"Content-Type":"application/json; charset=u
    res.end(msg);
  });
}
```

這裡的程式和前面 POST 範例，基本上大同小異，差別在：

- 幫user的資料加上id，隨意存放一些文字進去，讓Server回傳的資料多於Client端傳上來的，不然會覺得Server都沒做事。
- 增加了msg這個變數，存放將user物件JSON文字化的結果。JSON.stringify這個轉換函式是V8引擎所提供的，如果你好奇的話。
- 大重點來了，我們要告訴Client端，這次回傳的資料格式是JSON，所在Content-type和Content-Length要提供給Client。

Server很輕鬆就完成任務了，最後進行程式測試，啟動 Node.js 主程式之後，開啟瀏覽器就會看到表單，填寫完畢按下送出，就可以看到結果了。

最後 Node.js 本篇範例程式碼如下，

```
var http = require('http'),
    url  = require('url'),
    fs   = require("fs"),
    qs   = require('querystring'),
    server;

server = http.createServer(function (req,res) {
  var urlData,
      encode   = "utf8",
      filePath = "view/express_ajax_example_form.html",
      formData,
      action;

  urlData = url.parse(req.url,true);
  action = urlData.pathname;

  if (action === "/Signup") {
    formData = '';
    req.on("data", function (data) {

      formData += data;

    });

    req.on("end", function () {
      var msg;

      user = qs.parse(formData);
      user.id = "123456";
      msg = JSON.stringify(user);
      res.writeHead(200, {"Content-Type":"application/json;","Conte
      res.end(msg);
    });
  }
  else {
```

```
    fs.readFile(filePath, encode, function(err, file) {
      res.writeHead(200, {"Content-Type":"text/html; charset=utf-8'
      res.write(file);
      res.end();
    });
  }

});

server.listen(3000);

console.log('Server跑起來了，現在時間是:' + new Date());
```

原始資料提供

- [Node.JS初學者筆記(1)-用GET傳送資料] (http://ithelp.ithome.com.tw/question/10087402)
- [Node.JS初學者筆記(2)-用POST傳送資料] (http://ithelp.ithome.com.tw/question/10087489)
- [Node.JS初學者筆記(3)-用Ajax傳送資料] (http://ithelp.ithome.com.tw/question/10087627)

前言

在開始本章節以前先來談談MVC架構

M(MODEL)V(VIEW)C(CONTROLL)

MVC是近年來流行的web架構，但這個概念並不限定在web上，連android都有實做類似的概念。 何謂MVC?你問工程師每個人的看法都不大一樣，但他們都一致同意好的工程師一定要會MVC。

在這裡提供wiki看法:

（控制器 **Controller**）- 負責轉發請求，對請求進行處理。

（視圖 **View**） - 介面設計人員進行圖形介面設計。

（模型 **Model**） - 程式設計師編寫程式應有的功能（實作演算法等等）、資料庫專家進行資料管理和資料庫設計(可以實作具體的功能)。

轉換成Node.js的說法就是：

（控制器 **Controller**）- server(express)，接受參數(POST OR GET)後轉傳至要執行的程式，若有需要則回傳結果。

（視圖 **View**） - html部分，為了輸出Controller的訊息，會需要用到view engine，下面會介紹ejs跟angular.js

（模型 **Model**）- 邏輯處理

實例一 MVC基本範例

讓我們做個範例，建立一個名叫**node_mvc_1**目錄

你可以參考 https://github.com/y2468101216/node-wiki-gitbook/tree/master/src/node_mvc_1

結構:

```
node_mvc_1/
├── bin/
|   ├── download.js
├── download/
|   ├── a.txt
|   ├── b.txt
├── public/
|   ├── index.html
└── app.js
```

bin/download.js:

```
/**
 * Name:download.js
 * Purpose:Model download example
 * Author:Yun
 * Version:1.0
 * Update:2015-09-22
 */
module.exports = function(){
    this.checkFile = function(pathString, callback){
        var fs = require('fs');
        fs.stat(pathString,function(err, stats){
            if(!err){
                callback(true);
            }else{
                callback(false);
            }
        });
    }
}
```

public/index.html:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Node MVC 1</title>
</head>
<body>
    <div>
        <a href="./download/a.txt">download a</a>
    </div>
    <div>
        <a href="./download/b.txt">download b</a>
    </div>
    <div>
        <a href="./download/c.txt">download c</a>
    </div>
</body>
</html>
```

app.js:

```
/**
 * Name:app.js
 * Purpose:controller express example
 * Author:Yun
 * Version:1.0
 * Update:2015-09-22
 */

var express = require('express');
var app = express();

app.get('/', function(req, res) {
    var options = {
        root : __dirname + '/public/',
        dotfiles : 'deny',
        headers : {
            'x-timestamp' : Date.now(),
            'x-sent' : true
```

```
        }
    };

    res.sendFile('index.html', options, function(err) {
        if (err) {
            res.status(err.status).end();
        } else {
            // console.log('Sent:index.html');
        }
    });
});

app.get('/download/:name', function(req, res) {
    var downloadClass = require('./bin/download.js');
    var dl = new downloadClass();
    var pathString = './download/' + req.params.name;

    dl.checkFile(pathString, function(isFile) {
        if (isFile) {
            res.download(pathString);
        } else {
            res.send('error');
        }
    });
});

console.log('server is running');

app.listen(8080);
```

這是一個下載檔案的範例，在index.html裡有三個超連結分別對應a.txt、b.txt、c.txt。但你可以注意到在download目錄裡 並沒有c.txt，所以他應該會回傳error，不過今天的重點不在那邊，讓我們回頭來看app.js

app.js把檢查檔案是否存在的邏輯處理抽離另外做成一個class，讓app.js只單純處理controller的問題：轉傳給model，回傳給view。

這樣的好處是你易於維護，不會因為一個程式bug導致整個server crush，當然檔案跟程式碼會變得比較多，但整體而言複雜度是下降的。

實例二 輸出文字

在Node.js裡面view engine分成兩個：jade跟ejs，我會談跟php想法比較接近的ejs。

如果跟我一樣有做過php的話，那你應該記得php本身即是個view engine這件事，但很可惜的是Node.js並不能直接 這樣輸出，但他有類似的東西可以讓你無痛轉換，沒錯！就是ejs

讓我們做個範例，建立一個名叫**node_mvc_2_ejs**目錄

你可以參考 https://github.com/y2468101216/node-wiki-gitbook/tree/master/src/node_mvc_2_ejs

先安裝ejs:

```
npm install ejs
```

結構:

```
node_mvc_2_ejs/
├── bin/
|   ├── login.js
├── views/
|   ├── index.html
└── app.js
```

login.js:

```
/**
 * Name:login.js
 * Purpose:ejs login example
 * Author:Yun
 * Version:1.0
 * Update:2015-09-22
 */

module.exports = function (){
    this.check = function (account, password, callback){
        var message = 'login success';
        var error = false;

        if(account != 'test' && error == false){
            message = 'account error';
            error = true;

        }

        if(password != 'test' && error == false){
            message = 'password error';
            error = true;
        }

        callback(message, error);
    }
}
```

index.html:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Node MVC 2 EJS</title>
</head>
<body>
<form action="/login" method="post">
    <div>
        <div><label>Account:</label></div>
        <input type="text" name="account" />
    </div>
    <div>
        <div><label>Password:</label></div>
        <input type="password" name="password" />
    </div>
    <div>
        <button type="submit">submit</button>
    </div>
    <% if (message) { %>
        <% if (error) { %>
            <label style="color:red;"><%= message %></label>
        <% } else { %>
             <label style="color:green;"><%= message %></label>
         <% } %>
     <% } %>
</form>
</body>
</html>
```

app.js:

```
/**
 * Name:app.js
 * Purpose:ejs express example
 * Author:Yun
 * Version:1.0
 * Update:2015-09-22
 */
```

```
var express = require('express');
var bodyParser = require('body-parser');

var app = express();

//create application/x-www-form-urlencoded parser
app.use(bodyParser.urlencoded({
    extended : true
}));

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.engine('html', require('ejs').renderFile);

//index page
app.get('/', function(req, res) {
    res.render('index.html',{message:false});
});

//login
app.post('/login',function(req, res){
    var loginClass = require('./bin/login.js');
    var login = new loginClass();
    login.check(req.body.account, req.body.password,function(returr
        res.render('index.html',{message:returnMessage,error:isErrc
    });
});

console.log('server is running');

app.listen(8080);
```

Account:

Password:

submit

Account:

Password:

submit

account error

Account:

Password:

submit

password error

Account:

Password:

submit

login success

讓我們來討論一下app.js裡面多的東西

```
app.use(bodyParser.urlencoded({
    extended : true
}));
```

這段是說如果封包裡的body有東西的話，用qs library去爬，關於要採用qs library還是query string 請參考以下連結：http://stackoverflow.com/questions/29175465/body-parser-extended-option-qs-vs-querystring

```
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.engine('html', require('ejs').renderFile);
```

app.set就是可以設定一些express的常數，讓他知道要去哪找。

第一行設定views的目錄在哪。

第二行設定要使用的view engine。

第三行設定說，如果是html的話一樣送給ejs解釋。也就是說現在你在目錄擺html跟ejs，都會被掃過看有沒有ejs語法。

jade與ejs的優缺點

優點：

1. client端呈現速度快
2. 有學過view engine能夠快速上手

缺點：

1. server端要耗用資源
2. 版面醜陋
3. 做假資料時難度會提升很多
4. 對前端工程師不友善
5. 要多學一種語言

另一個選擇-使用client side js framework:angular.js or react.js

有鑒於view engine的眾多缺點，逐漸有部分人改採用angular.js跟react.js做為view engine的替代品。

讓我們做個範例，建立一個名叫**node_mvc_3_angularjs**目錄

你可以參考 https://github.com/y2468101216/node-wiki-gitbook/tree/master/src/node_mvc_3_angularjs

結構：

```
node_mvc_3_angularjs/
├── bin/
|   ├── login.js
├── public/
|   ├── index.html
└── app.js
```

login.js:

```
/**
 * Name:login.js
 * Purpose:ejs login example
 * Author:Yun
 * Version:1.0
 * Update:2015-09-22
 */

module.exports = function (){
    this.check = function (account, password, callback){
        var message = 'login success';
        var error = false;

        if(account != 'test' && error == false){
            message = 'account error';
            error = true;
        }

        if(password != 'test' && error == false){
            message = 'password error';
            error = true;
        }

        callback(message, error);
    }
}
```

index.html:

```
<!DOCTYPE html>
<html ng-app="node_mvc_app">
<head>
<meta charset="UTF-8">
<title>Node MVC 3 angularjs</title>
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootst
<script src="https://code.jquery.com/jquery-2.1.4.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.4.
</head>
<body ng-controller="node_mvc_controller">
```

```
<form>
    <div>
        <div><label>Account:</label></div>
        <input type="text" name="account" ng-model="user.account" /
    </div>
    <div>
        <div><label>Password:</label></div>
        <input type="password" name="password" ng-model="user.passw
    </div>
    <div>
        <button type="button" ng-click="loginCheck()">submit</butto
    </div>

</form>
<label ng-style="messageStyle">{{message}}</label>
</body>
<script type="text/javascript">
var app = angular.module('node_mvc_app', []);
app.controller('node_mvc_controller',function($scope, $http){
    $scope.loginCheck = function () {
        var req = {
            method: 'POST',
            url: 'login',
            headers: {
                'Content-Type': 'application/x-www-form-urlencoded,
            },
            data: $.param($scope.user)
        }
        $http(req).success(function(data){
            if(data.error){
                $scope.messageStyle = {color:'red'};
            }else{
                $scope.messageStyle = {color:'green'};
            }
            $scope.message = data.message;
        });
    }
});
</script>
</html>
```

app.js:

```
/**
 * Name:app.js
 * Purpose:ejs express example
 * Author:Yun
 * Version:1.0
 * Update:2015-09-22
 */

var express = require('express');
var bodyParser = require('body-parser');

var app = express();

var options = {
        root : __dirname + '/public/',
        dotfiles : 'deny',
        headers : {
            'x-timestamp' : Date.now(),
            'x-sent' : true
        }
    };

//create application/x-www-form-urlencoded parser
app.use(bodyParser.urlencoded({
    extended : true
}));

//index page
app.get('/', function(req, res) {
    res.sendFile('index.html',options,function(err){
        if(err){
            console.log(err);
        }
    });
});
```

```
//login
app.post('/login',function(req, res){
    var loginClass = require('./bin/login.js');
    var login = new loginClass();
    login.check(req.body.account, req.body.password,function(returr
        res.send({message:returnMessage,error:isError});
    });
});

console.log('server is running');

app.listen(8080);
```

Account:

Password:

submit

Account:

test

Password:

submit

account error

Account:

test

Password:

submit

password error

Account:

test

Password:

••••

submit

login success

login.js本身並沒有變動，所以我們直接來看index.html

這邊我們改用ajax去取得是否有登入成功。ajax會給使用者比較好的體驗，不然反覆刷新頁面使用者也很煩， 而且資料不用多做設定就可以一直留在網頁上，關於angular.js的語法可以參考官網，這邊就不再多做說明。

app.js因為我們不再使用ejs了，所以我們將app.set的部分全拿掉，這也導致res.render必須拿掉。 所以我們改成res.sendFile。

```
res.sendFile('index.html',options,function(err){
    if(err){
        console.log(err);
    }
});
```

回傳的部分，server丟json出來是最好的，因為JavaScript對json的支援很好，而且在提供相同資訊的情況下 json遠比xml來的簡單。

```
app.post('/login',function(req, res){
    var loginClass = require('./bin/login.js');
    var login = new loginClass();
    login.check(req.body.account, req.body.password,function(returr
        res.send({message:returnMessage,error:isError});
    });
});
```

client side js framework的優缺點

優點：

1. server耗用資源較少。
2. 版面整潔
3. 對前端友善
4. 做假資料時比較輕鬆

缺點：

1. include page會十分緩慢
2. 必定要先載入該framework
3. 使用者多時ajax可能會對server造成更多負擔

結語

如上面所敘，兩種方法都有優缺點，開發時應該依照專案需求去做選擇，甚至可以兩種混用，不要被本篇文章侷限住了。

參考資料

- wiki-MVC:https://zh.wikipedia.org/wiki/MVC
- angular.js:https://angularjs.org
- stackoverflow-

qs_vs_querystring:http://stackoverflow.com/questions/29175465/body-parser-extended-option-qs-vs-querystring

前言

不懂Database的人不是Backend Engineer，可見Database多麼重要。 本章節將會舉一個關聯式資料庫:mysql跟一個NOSQL:MongoDB的串連。 本章節不會撰述安裝資料庫跟SQL語法

關聯式資料庫:mysql

首先須先安裝mysql套件:

```
npm install mysql
```

範例一:基本範例

https://github.com/y2468101216/node-wiki-gitbook/tree/master/src/node_mysql/mysql.js

```
/**
 * Name:mysql.js
 * Purpose:mysql 連接教學
 * Author:Yun
 * Version:1.0
 * Update:2015-09-25
 */

var mysql = require('mysql');

//建立連線資料
var connection = mysql.createConnection({
  host     : 'localhost',//資料庫IP
  user     : 'root',//使用者名稱
  password : 'root',//使用者密碼
  database : 'localhost'//資料庫名稱
});

//嘗試連線
connection.connect();

//倒出SQL結果
connection.query('SELECT 1 + 1 AS solution', function(err, rows, fi
  if (err) throw err;

  console.log('The solution is: ', rows[0].solution);
});

//結束連線
connection.end();
```

結果:

```
The solution is:  2
```

比較值得一提的是 `connection.end();` 這個語法，如果你有寫php就知道PDO是不需要close connection的。 (詳見 http://stackoverflow.com/questions/18277233/pdo-closing-connection)

也就是說他比較像mysqli。

範例二:Parameterized Query

在開始之前先講一下SQL Injection:

- wiki的解釋:

SQL攻擊（SQL injection），簡稱隱碼攻擊，是發生於應用程式之資料庫層的安全漏洞。簡而言之，是在輸入的字串之中夾帶SQL指令，在設計不良的程式當中忽略了檢查，那麼這些夾帶進去的指令就會被資料庫伺服器誤認為是正常的SQL指令而執行，因此遭到破壞或是入侵。

所以就出現了Parameterized Query：

- wiki的解釋:

參數化查詢（Parameterized Query或Parameterized Statement）是指在設計與資料庫連結並存取資料時，在需要填入數值或資料的地方，使用參數（Parameter）來給值，這個方法目前已被視為最有效可預防SQL資料隱碼攻擊的攻擊手法的防禦方式。

除了安全因素，相比起拼接字串的SQL語句，參數化的查詢往往有效能優勢。因為參數化的查詢能讓不同的資料通過參數到達資料庫，從而公用同一條SQL語句。大多數資料庫會快取解釋SQL語句產生的位元組碼而省下重複解析的開銷。如果採取拼接字串的SQL語句，則會由於運算元據是SQL語句的一部分而非參數的一部分，而反覆大量解釋SQL語句產生不必要的開銷。

https://github.com/y2468101216/node-wiki-gitbook/tree/master/src/node_mysql/node_mysql_parameterized_query.js

```
/**
 * Name:mysql.js
 * Purpose:mysql Parameterized Query
 * Author:Yun
 * Version:1.0
 * Update:2015-09-25
 */

var mysql = require('mysql');

var connection = mysql.createConnection({
  host     : 'localhost',//資料庫IP
  user     : 'root',//使用者名稱
  password : 'root',//使用者密碼
  database : 'localhost'//資料庫名稱
});

//嘗試連線
connection.connect();

//倒出SQL結果
connection.query('SELECT ? + ? AS solution',[1,2], function(err, ro
  if (err) throw err;

  console.log('The solution is: ', rows[0].solution);
});

//結束連線
connection.end();
```

你可以注意到只有 `connection.query` 改變而已，裡面多插入一個陣列，這種寫法在需要大量插入或更新重複的SQL時異常好用， 你不需要撰寫SQL只需更換參數就可以，可以節省許多行數。

NOSQL:MongoDB

NOSQL是最近很火紅的資料庫型態，特徵是不使用任何SQL語言、不需要規劃table架構，是一個新興的資料庫型態。 我會花比較多篇幅講這個，因為這是一個從觀念上完全不一樣的東西。

首先須先安裝mongodb套件:

```
npm install mongodb
```

- 插入資料https://github.com/y2468101216/node-wiki-gitbook/tree/master/src/node_mongodb/mongodb_insert.js

```
/**
 * Name:mongodb_update.js
 * Purpose:connect & insert mongodb
 * Author:Yun
 * Version:1.0
 * Update:2015-09-30
 */

var MongoClient = require('mongodb').MongoClient;// mongodb client
var assert = require('assert');// 測試工具

var url = 'mongodb://localhost:27017/test';// mongodb://登入url/db名

//插入資料到
var insertDocument = function(db, callback) {
    // 打開集合 (沒有的話會自動建一個)->插入一筆資料
    db.collection('restaurants').insertOne({
        "address" : {
            "street" : "2 Avenue",
            "zipcode" : "10075",
            "building" : "1480",
            "coord" : [ -73.9557413, 40.7720266 ]
        },
        "borough" : "Manhattan",
        "cuisine" : "Italian",
        "grades" : [ {
            "date" : new Date("2014-10-01T00:00:00Z"),
            "grade" : "A",
```

```
                "score" : 11
            }, {
                "date" : new Date("2014-01-16T00:00:00Z"),
                "grade" : "B",
                "score" : 17
            } ],
            "name" : "Vella",
            "restaurant_id" : "41704620"
        }, function(err, result) {
            assert.equal(err, null);// 如果不是期望值(null), 則throw error
            console.log("Inserted a document into the restaurants colle
            callback(result);
        });
};

//進行連線
MongoClient.connect(url, function(err, db) {
    assert.equal(null, err);// 如果不是期望值(null), 則throw error
    insertDocument(db, function() {
        db.close();//關閉連線
    });
});
```

執行後印出

```
Inserted a document into the restaurants collection.
```

你可以注意到在存資料時他的擴展性很強(注意grades)，不同於一般的資料庫。

- 查詢資料https://github.com/y2468101216/node-wiki-gitbook/tree/master/src/node_mongodb/mongodb_query.js

```
/**
 * Name:mongodb_query.js
 * Purpose:connect & insert mongodb
 * Author:Yun
 * Version:1.0
 * Update:2015-10-01
```

```
 */

var MongoClient = require('mongodb').MongoClient;//mongodb client
var assert = require('assert');// 測試工具
var url = 'mongodb://localhost:27017/test';// mongodb://登入url/db名

//查詢資料
var findRestaurants = function(findCondition, db, callback) {
    var cursor =db.collection('restaurants').find(findCondition);
    //將每筆資料倒出來
    cursor.each(function(err, doc) {
    assert.equal(err, null);
    if (doc != null) {
         //列印查詢條件
         console.dir('find:');
         console.log(findCondition);
         //列印資料
      console.dir(doc);
    } else {
      callback();
    }
    });
};

//列出全部的集合裡的資料
MongoClient.connect(url, function(err, db) {
  assert.equal(null, err);
  findRestaurants(null, db, function() {
    db.close();
  });
});

//尋找address.zipcode等於10075的
MongoClient.connect(url, function(err, db) {
  assert.equal(null, err);
  findRestaurants({ "address.zipcode": "10075" }, db, function() {
    db.close();
  });
});
```

```
//尋找address.zipcode等於10076的
MongoClient.connect(url, function(err, db) {
  assert.equal(null, err);
  findRestaurants({ "address.zipcode": "10076" }, db, function() {
    db.close();
  });
});
```

執行結果:

```
'find:'

{ _id: { _bsontype: 'ObjectID', id: 'V\fß¨8ßZ×Bo;\\' },
  address:
   { street: '2 Avenue',
     zipcode: '10075',
     building: '1480',
     coord: [ -73.9557413, 40.7720266 ] },
  borough: 'Manhattan',
  cuisine: 'Italian',
  grades:
   [ { date: Wed Oct 01 2014 08:00:00 GMT+0800 (CST),
       grade: 'A',
       score: 11 },
     { date: Thu Jan 16 2014 08:00:00 GMT+0800 (CST),
       grade: 'B',
       score: 17 } ],
  name: 'Vella',
  restaurant_id: '41704620' }
'find:'
{ 'address.zipcode': '10075' }
{ _id: { _bsontype: 'ObjectID', id: 'V\fß¨8ßZ×Bo;\\' },
address:
 { street: '2 Avenue',
   zipcode: '10075',
   building: '1480',
   coord: [ -73.9557413, 40.7720266 ] },
borough: 'Manhattan',
cuisine: 'Italian',
```

```
grades:
[ { date: Wed Oct 01 2014 08:00:00 GMT+0800 (CST),
    grade: 'A',
    score: 11 },
  { date: Thu Jan 16 2014 08:00:00 GMT+0800 (CST),
    grade: 'B',
    score: 17 } ],
name: 'Vella',
restaurant_id: '41704620' }

'find:'
{ 'address.zipcode': '10076' }
```

你可以注意到mongodb插入的時候多插了一個_id，那是mongodb的主鍵，不重複唯一， mongodb用它來分別每一筆資料。

find其實就相當SQL裡的where，但他比where強的地方是在於說他可以查詢dot notation(EX:address.zipcode)。 這點是SQL所做不到的

- 更新資料https://github.com/y2468101216/node-wiki-gitbook/tree/master/src/node_mongodb/mongodb_query.js

```
/**
 * Name:mongodb_update.js
 * Purpose:connect & insert mongodb
 * Author:Yun
 * Version:1.0
 * Update:2015-10-02
 */

var MongoClient = require('mongodb').MongoClient;// mongodb client
var assert = require('assert');// 測試工具

var url = 'mongodb://localhost:27017/test';// mongodb://登入url/db名

var updateRestaurants = function(db, callback) {
    db.collection('restaurants').updateOne(
      { "name" : "Vella" },//設定條件
      {
        $set: { "cuisine": "American (New)" },
        $currentDate: { "lastModified": true }
      }, //設定更新項目
      function(err, results) {
      console.log(results);//印出更新結果
      callback();
    });
};

MongoClient.connect(url, function(err, db) {
  assert.equal(null, err);

  updateRestaurants(db, function() {
    db.close();
  });
});
```

執行結果:

```
{ result: { ok: 1, nModified: 1, n: 1 },
  connection:
```

```
    { domain: null,
      _events:
       { close: [Object],
         error: [Object],
         timeout: [Object],
         parseError: [Object],
         connect: [Function] },
      _maxListeners: undefined,
      options:
       { socketOptions: {},
         auto_reconnect: true,
         host: 'localhost',
         port: 27017,
         cursorFactory: [Object],
         reconnect: true,
         emitError: true,
         size: 5,
         disconnectHandler: [Object],
         bson: {},
         messageHandler: [Function],
         wireProtocolHandler: {} },
      id: 2,
      logger: { className: 'Connection' },
      bson: {},
      tag: undefined,
      messageHandler: [Function],
      maxBsonMessageSize: 67108864,
      port: 27017,
      host: 'localhost',
      keepAlive: true,
      keepAliveInitialDelay: 0,
      noDelay: true,
      connectionTimeout: 0,
      socketTimeout: 0,
      destroyed: false,
      domainSocket: false,
      singleBufferSerializtion: true,
      serializationFunction: 'toBinUnified',
      ca: null,
      cert: null,
```

```
      key: null,
      passphrase: null,
      ssl: false,
      rejectUnauthorized: false,
      responseOptions: { promoteLongs: true },
      flushing: false,
      queue: [],
      connection:
       { _connecting: false,
         _hadError: false,
         _handle: [Object],
         _parent: null,
         _host: 'localhost',
         _readableState: [Object],
         readable: true,
         domain: null,
         _events: [Object],
         _maxListeners: undefined,
         _writableState: [Object],
         writable: true,
         allowHalfOpen: false,
         destroyed: false,
         bytesRead: 71,
         _bytesDispatched: 223,
         _pendingData: null,
         _pendingEncoding: '',
         _idleNext: null,
         _idlePrev: null,
         _idleTimeout: -1,
         read: [Function],
         _consuming: true },
      writeStream: null,
      buffer: null,
      sizeOfMessage: 0,
      bytesRead: 0,
      stubBuffer: null },
   matchedCount: 1,
   modifiedCount: 1,
   upsertedId: null,
   upsertedCount: 0 }
```

這樣只會更新第一筆找到的資料。 跟查詢一樣，你可以使用dot notation作為更新條件

- 刪除資料

```
/**
 * Name:mongodb_delete.js
 * Purpose:connect & update mongodb
 * Author:Yun
 * Version:1.0
 * Update:2015-10-02
 */

var MongoClient = require('mongodb').MongoClient;// mongodb client
var assert = require('assert');// 測試工具

var url = 'mongodb://localhost:27017/test';// mongodb://登入url/db名

var removeRestaurants = function(db, callback) {
    db.collection('restaurants').deleteOne(
      { "borough": "Queens" },//設定條件
      function(err, results) {
            console.log(results);//印出更新結果
            callback();
      }
    );
};

MongoClient.connect(url, function(err, db) {
    assert.equal(null, err);

    removeRestaurants(db, function() {
      db.close();
    });
});
```

```
{ result: { ok: 1, nModified: 1, n: 1 },
```

```
   connection:
    { domain: null,
      _events:
       { close: [Object],
         error: [Object],
         timeout: [Object],
         parseError: [Object],
         connect: [Function] },
      _maxListeners: undefined,
      options:
       { socketOptions: {},
         auto_reconnect: true,
         host: 'localhost',
         port: 27017,
         cursorFactory: [Object],
         reconnect: true,
         emitError: true,
         size: 5,
         disconnectHandler: [Object],
         bson: {},
         messageHandler: [Function],
         wireProtocolHandler: {} },
      id: 2,
      logger: { className: 'Connection' },
      bson: {},
      tag: undefined,
      messageHandler: [Function],
      maxBsonMessageSize: 67108864,
      port: 27017,
      host: 'localhost',
      keepAlive: true,
      keepAliveInitialDelay: 0,
      noDelay: true,
      connectionTimeout: 0,
      socketTimeout: 0,
      destroyed: false,
      domainSocket: false,
      singleBufferSerializtion: true,
      serializationFunction: 'toBinUnified',
      ca: null,
```

```
      cert: null,
      key: null,
      passphrase: null,
      ssl: false,
      rejectUnauthorized: false,
      responseOptions: { promoteLongs: true },
      flushing: false,
      queue: [],
      connection:
       { _connecting: false,
         _hadError: false,
         _handle: [Object],
         _parent: null,
         _host: 'localhost',
         _readableState: [Object],
         readable: true,
         domain: null,
         _events: [Object],
         _maxListeners: undefined,
         _writableState: [Object],
         writable: true,
         allowHalfOpen: false,
         destroyed: false,
         bytesRead: 71,
         _bytesDispatched: 223,
         _pendingData: null,
         _pendingEncoding: '',
         _idleNext: null,
         _idlePrev: null,
         _idleTimeout: -1,
         read: [Function],
         _consuming: true },
      writeStream: null,
      buffer: null,
      sizeOfMessage: 0,
      bytesRead: 0,
      stubBuffer: null },
   matchedCount: 1,
   modifiedCount: 1,
   upsertedId: null,
```

```
    upsertedCount: 0 }
```

這樣只會刪除一筆，跟update其實沒差多少

結語

MongoDB跟Node.js就跟mysql之於php一樣，天生一對，但這並不代表你不能用其他的資料庫。 根據專案選擇你要的才是正確的。

參考資料

- wiki-SQL injection:https://zh.wikipedia.org/wiki/SQL資料隱碼攻擊
- wiki-Parameterized Query:https://zh.wikipedia.org/wiki/參數化查詢
- node-mysql:https://github.com/felixge/node-mysql/#preparing-queries
- NOSQL:https://zh.wikipedia.org/wiki/NoSQL
- MONGODB:https://docs.mongodb.org/getting-started/node/

前言

test是程式中很重要的一環，本篇我們將會介紹如何用mocha.js、nightwatch.js、cucumber.js撰寫test case。

原始碼

https://github.com/y2468101216/node-wiki-gitbook/tree/master/src/node_test

測試方法的層級

1. 單元測試
2. 整合測試
3. 使用者測試

單元測試最易除錯，但不貼近使用行為，使用者測試則相反。

為何我們要使用TDD，或者說寫測試有何好處

TDD是一種用測試來進行開發的模式，所以他的本質其實是為了開發而非測試。

Kent Beck(設計模式的先驅者)在RIP TDD裡舉出了8個你應該使用TDD的理由。

1. Over-engineering(過度設計):

EX: 今天你被授命要做一個會員登入的系統，你老闆只要你串facebook登入，結果你多寫了一個google登入。 這樣就過度設計了，程式碼裡不要擺用不到的東西，會造成後面維護上的困擾。

TDD每一個測試都是需求，而你不應該寫需求以外的程式，TDD力求以最簡單的方法讓測試通過。

1. API feedback(介面回饋):

因為TDD會根據使用者的需求寫測試，當你發現 你的介面不敷使用於測試時，就會去修改介面，這會使你的介面越來越貼近使用者。

1. Logic errors(邏輯錯誤):

TDD裡面不會有任何的邏輯(if else)判斷，所以如果出來的結果不符合就是你的method有問題。 而且TDD一次只會有一個測試失敗，所以一定是你剛增加的code有問題。

1. Documentation(文件):

每個工程師都會跟你說他討厭程式沒有文件，但實際上會寫文件的很少，後面會繼續維護的更少了。

TDD的測試即文件，當你看完測試你就會瞭解這隻程式怎用了。 而且如果需求改變，你的測試也會改變，就會很自然地維護它了。

1. Feeling overwhelmed:

標題無關。

TDD的宗旨是先寫測試在開發，意味著即使沒有程式依然可以先寫測試，

1. Separate interface from implementation thinking(從邏輯來實踐獨立介面):

EX: 今天有個需求是串金流API，但是開發API的人說他要等上線前10天才能給你測試。

TDD遇到這種問題時就會做一個介面，測試時實作這個介面，去模擬API的行為。這樣你就不用因為別人拖延自己的進度。

1. Agreement(同意)

當你把需求解掉了以後，你要如何說服發出需求的人妳已經把問題解決掉了？ 顯然用測試是一個好方法。

1. Anxiety(焦慮)

當老闆問你一切是否OK時，TDD可以不用讓你提心吊膽的說OK。

安裝mocha.js

使用npm安裝

```
$ npm install -g mocha
```

mocha.js常用語法

before:全部測試開始之前先執行

it:測試案例

after:全部測試結束以後執行

beforeEach:每個測試前先執行

afterEach:每個測試後先執行

mocha.js 的第一個測試案例

firsrTest.js

```
var a;
var b;
var assert = require('assert');

describe('firstTest', function () {
    before(function () {
        a = 1;
        b = 2;
    });

    it('a + b should be 3', function () {
        assert.equal(a + b, 3);
    });
});
```

切換到該檔案目錄底下執行:

```
$ mocha firsrTest.js
```

```
firstTest
  ✓ a + b should be 3


1 passing (9ms)
```

—

比較object & array

arrayTest.js

```
var actual;
var exected;
var assert = require('assert');

describe('arrayTest', function () {
    before(function () {
        actual = [1, 2, 3];
        exected = [1, 2, 3];
    });

    it('[1,2,3] equal [1,2,3]', function () {
        assert.equal(actual, exected);
    });

    it('[1,2,3] deepequal [1,2,3]', function () {
        assert.deepEqual(actual, exected);
    });
});
```

第一個是一定不會過的，因為他們是兩個不同的array。 要比較這兩種請用deepEqual。

- 附註

```
actual = [1,2,3];
exected = actual;
```

這樣的話equal就會過，因為兩個是一樣的object了。

async

asyncTest.js

```
var assert = require('assert');

describe('asyncTest', function () {

    it('async_Test_Without_Done()', function () {
        var actual;
        var exected;
        var fs = require('fs');
        fs.stat('./asyncTest.js', function (err, stats) {
            var actual = stats.isFile();
            var exected = false;
            assert.equal(actual, exected);
        });
    });

    it('async_Test_With_Done()', function (done) {
        var actual;
        var exected;
        var fs = require('fs');
        fs.stat('./asyncTest.js', function (err, stats) {
            var actual = stats.isFile();
            var exected = false;
            assert.equal(actual, exected);
            done();
        });
    });
});
```

因為javascript的async特性，所以在做這種async的操作要記得放done()，確保不會立即回傳測試結果。

外部依賴

有的時候你想測試的function必須用到外面的class，而那個class還沒寫好怎辦？ 最常見的就是ORM還沒寫好，但你需要串資料庫。

沒關係我們可以模擬該class的行為，假裝有那個class的存在。

interfaceTest.js:

```
var dbConn = {};
var assert = require('assert');
var order = require('./bin/order.js');
var orderTest = new order();


describe('firstTest', function () {
    before(function () {
        dbConn.Memberselect = function (memberName) {
            if (typeof memberName == 'string') {
                if (memberName == 'a') {
                    var cursor = { 1: { name: 'a', price: 100, numb
                } else {
                    var cursor = null;
                }
            } else {
                var cursor = null;
            }
            return cursor;
        }
    });

    it('price total should be 500', function () {
        var cursor = dbConn.Memberselect('a');
        orderTest.setOrder(cursor);
        var expected = 500;
        var actual = orderTest.priceTotal();
        assert.equal(actual, expected);
    });

    it('price total should be 0', function () {
        var cursor = dbConn.Memberselect(null);
        orderTest.setOrder(cursor);
        var expected = 500;
        var actual = orderTest.priceTotal();
        assert.equal(actual, expected);
    });
});
```

我設計了一個計算訂單總金額的功能，這個訂單會從DB裡面被撈出來，但ORM還沒寫好。 於是我模擬了ORM的行為去避免這個問題。

nightwatch.js簡介(整合測試)

這是一個測試end to end的好工具，她與Selenium結合使其可以自動打開browser做end to end測試，一般而言他會歸在整合測試中，但他也可以拿來做使用者測試。

使用時機

1. 當你想要重構某支程式，而他並不適合寫unit test
2. 給PM或者那些不懂程式的人看你的程式是如何運行的
3. 想要測試流程時，比如登入行為
4. 想要測試前端

安裝nightwatch.js

跟mocha一樣，我們希望它可以可以在系統的任何地方run。

```
$ npm install nightwatch
```

Selenium安裝

Selenium需要java支援，請先確定你的PC上有java

然後我們需要下載Seleniumhttp://selenium-release.storage.googleapis.com/index.html 請點選網址後下載最新版的

- option

Selenium本身就內建支援firefox，但是如果你想支援chrome的話就要另外下載chromedriver。 請至https://sites.google.com/a/chromium.org/chromedriver/downloads下載最新版

設定環境

這邊我只挑基本常用的的出來講，其餘可以看nightwatch.js官網裡的doc

請先新增一個目錄，內容如下

```
nightwatch
nightwatch.json
libs/
  ├── selenium-server-standalone.jar
  ├── chromedriver
reports/
screenshots/
tests/
  └── search
      └── googleSearchTest.js
```

在你的測試專案根目錄新增nightwatch.json:

```
{
  "src_folders" : ["tests"],//你的測試檔案目錄
  "output_folder" : "reports",//如果要輸出報告時，輸出的目錄
  "custom_commands_path" : "",
  "custom_assertions_path" : "",
  "page_objects_path" : "",
  "globals_path" : "",

  //selenium設定
  "selenium" : {
    "start_process" : false, //是否自動啟動Selenium
    "server_path" : "",//Selenium jar位置
    "log_path" : "",//輸出Selenium的log位置
    "host" : "127.0.0.1",//設定Selenium 伺服器IP
    "port" : 4444,//設定Selenium 伺服器PORT
    "cli_args" : {
      "webdriver.chrome.driver" : "",//chromedriver位置
      "webdriver.ie.driver" : ""//IE eats shit
    }
  },
```

```
  //測試設定
  "test_settings" : {
    "default" : {
      "launch_url" : "http://localhost",
      "selenium_port"  : 4444,//連結Selenium 伺服器PORT
      "selenium_host"  : "localhost",//連結Selenium 伺服器網址
      "silent": true,
      "screenshots" : {
        "enabled" : true,//是否拍照
          "on_failure" : true,//測試失敗時拍照
         "on_error" : false,//指令錯誤時拍照
          "path" : ""//拍照路徑
      },
      //預設啟動的browser
      "desiredCapabilities": {
        "browserName": "firefox",//預設啟動的browser
        "javascriptEnabled": true,
        "acceptSslCerts": true
      }
    },

    //自定義browser, 之後使用nightwatch可能會用到
    "chrome" : {
      "desiredCapabilities": {
        "browserName": "chrome",//預設啟動的browser
        "javascriptEnabled": true,
        "acceptSslCerts": true
      }
    }
  }
}
```

- option

如果你想要自動啟動Selenium的話請更改你的nightwatch.json為

```
start_process : true,
server_path : "/你的目錄/selenium-server-standalone-{VERSION}.jar"
```

linux&mac:

新增一個nightwatch檔案在專案根目錄底下，內容如下

```
#!/usr/bin/env node
require('nightwatch/bin/runner.js');
```

把它設定為可執行

```
$ chmod a+x nightwatch
```

windows:

新增一個nightwatch.js檔案在專案根目錄底下，內容如下

```
require('nightwatch/bin/runner.js');
```

用node先跑起來

```
> node nightwatch.js
```

nightwatch.js的第一個測試

search/googleSearch.js:

```
module.exports = {
    'search google test':function(browser){
        browser
        .url('http://www.google.com.tw')
        .waitForElementVisible('body', 1000)
        .setValue('input[type=text]', 'google')
        .keys(browser.Keys.ENTER)
        .pause(1000)
        .assert.containsText("ol#rso a", "Google")
        .end();
    }
}
```

- option

如果你並沒有設定Selenium自動執行，請先手動執行

```
$ java -jar selenium-server-standalone-{VERSION}.jar
```

進行測試，測試前請先切換到專案根目錄：

```
$ nightwatch tests/search/googleSearchTest.js
```

```
Starting selenium server... started - PID:  8384

[Search / Google Search Test] Test Suite
========================================

Running:  search google test
 ✔ Element <body> was visible after 120 milliseconds.
 ✔ Testing if element <ol#rso a> contains text: "Google".

OK. 2 assertions passed. (6.539s)
```

TDD 與 BDD 的差別

在TDD的宗旨：需求即測試、測試即開發，理論上TDD應該也可以讓PM跟SA加入。不然需求定義不清的情況下，你也沒辦法使用TDD。

但是這裡有一個問題：TDD大概只有工程師看得懂(node.js比較沒有這問題，因為他底下的測試工具幾乎都允許你用BDD模式開發)。 為了讓PM跟SA也能看懂並且修正需求，BDD就這樣橫空出世。

BDD 的代言人:cucumber

cucumber原本是for ruby的測試工具，但是因為他裡面的設計模式十分不錯，被轉成許多語言(JAVA、C#、JAVASCRIPT、PHP)等。 他運用了簡單的幾個單字，讓工程師與PM更易於釐清需求。

cucumber的好處

1. 程式碼與需求分開，不會不小心改到測試程式
2. 測試程式看起來更人性化。
3. 關鍵字足夠適用於各類需求

cucumber單字簡介

下面將會介紹幾個cucumber的常用單字

- Feature:產品名稱

EX:要開發的是購物車那就會寫上Feature:shoppingCar

- Background:

他會在before之後的每個Scenario開始以前執行一次就像是mocha的beforeEach

- Scenario:功能名稱

EX:將商品放入購物車那就會寫上Scenario:put item in shoppingCar

- Given:帶入參數

- When:運算得到結果

- Then:比對結果跟預期的是否一樣。

cucumber.js 安裝

跟mocha一樣，我們希望它可以可以在系統的任何地方run。

```
$ npm install -g cucumber
```

cucumber example

我們要做一個shopingCar。

建立一個目錄如下

```
features/
  ├── step_definitions
  |         └── shoppingCarStep.js
  ├── support
  |         ├──    hook.js
  |         └──    world.js
  └── shoppingCar.feature
lib/
  └── shoppingCar.js
package.json
```

以下為目錄解析

- features:是擺你的測試案例
- lib:是擺你要測試的module
- step_definitions:是擺測試步驟
- support:擺測試前後要做的程式

遵循TDD的原則一次只寫一個測試，打開features/shoppingCar.feature，撰寫內容如下:

```
Feature: shoppingCar

  Scenario: calculate apple price
    Given the item "apple"
    And the numbers "4"
    When the calculator is run
    Then the output should be "200"
```

features/step_definitions/shoppingCarStep.js:

```
/**
 * calculate step
 */

module.exports = function() {
    var self = this;

    this.Given('the item "$itemName"', function(itemName, callback)
        self.itemName = itemName;
        callback();
    });

    this.Given('the numbers "$numbers"', function(numbers, callback
        self.numbers = numbers;
        callback();
    });

    this.When(/^the calculator is run$/, function(callback) {
        self.result = self.calculator.priceCal(self.itemName, self.
        callback();
    });

    this.Then('the output should be "$output"', function(output, ca
        self.assert.equal(self.result,output);
        callback();
    });
}
```

讓我們來測試一下，請先切換到專案根目錄底下

```
$ cucumber.js features/shoppingCar.feature
```

```
Feature: shoppingCar

  Scenario: calculate apple price   # features/shoppingCar.feature:3
    Given the item "apple"          # features/shoppingCar.feature:4
    And the numbers "4"             # features/shoppingCar.feature:5
    When the calculator is run      # features/shoppingCar.feature:6
      TypeError: Cannot read property 'priceCal' of undefined
          at World.<anonymous> (/Users/Yun/github_nodejs/node-wiki-gitbook/src/n
ode_test/cucumber/features/step_definitions/shoppingCarStep.js:19:38)
    Then the output should be "200" # features/shoppingCar.feature:7

Failing scenarios:
features/shoppingCar.feature:3 # Scenario: calculate apple price

1 scenario (1 failed)
4 steps (1 failed, 1 skipped, 2 passed)
0m00.004s
```

當然會失敗，因為我們還沒有撰寫程式。

在撰寫程式前，我們注意到需求裡並沒有給我們價錢，

在實際工作上你應當要詢問每個蘋果的價錢是否是一樣的，不過這裡我們當作每顆蘋果的價錢是一樣的。

lib/shoppingCar.js:

```
/**
 * @function
 *
 * simple shoppingCar
 */
var shoppingCar = module.exports = function () {
    var fruitPrice = { apple: 50 };

    /**
    * simple calculate implementation
    * @param String name an fruit's name
    * @param int numbers how many fruit
    * @return int totalPrice
    */
    this.priceCal = function (name, numbers) {
        return fruitPrice[name] * numbers;
    }
};
```

features/support/world.js

```
/**
 * @function
 *
 * world is a constructor function
 * with utility properties,
 * destined to be used in step definitions
 */
var cwd = process.cwd();
var path = require('path');

var Calculator = require(path.join(cwd, 'lib', 'shoppingCar'));

module.exports = function() {
    this.calculator = new Calculator();
    this.assert = require('assert');
}
```

```
Feature: shoppingCar

  Scenario: calculate apple price    # features/shoppingCar.feature:3
    Given the item "apple"           # features/shoppingCar.feature:4
    And the numbers "4"              # features/shoppingCar.feature:5
    When the calculator is run       # features/shoppingCar.feature:6
    Then the output should be "200" # features/shoppingCar.feature:7

1 scenario (1 passed)
4 steps (4 passed)
0m00.003s
```

此時我們增加了第二個需求。

```
  Scenario: calculate orange price
     Given the item "orange"
     And the numbers "3"
     When the calculator is run
     Then the output should be "120"
```

多了orange，當然還是沒寫價錢，所以我們假定他一顆40元

修改lib/shoppingCar.js:

```
var fruitPrice = { apple: 50, orange: 40 };
```

```
Feature: shoppingCar

  Scenario: calculate apple price    # features/shoppingCar.feature:3
    Given the item "apple"           # features/shoppingCar.feature:4
    And the numbers "4"              # features/shoppingCar.feature:5
    When the calculator is run       # features/shoppingCar.feature:6
    Then the output should be "200" # features/shoppingCar.feature:7

  Scenario: calculate orange price    # features/shoppingCar.feature:9
    Given the item "orange"           # features/shoppingCar.feature:10
    And the numbers "3"               # features/shoppingCar.feature:11
    When the calculator is run        # features/shoppingCar.feature:12
    Then the output should be "120"  # features/shoppingCar.feature:13

2 scenarios (2 passed)
8 steps (8 passed)
0m00.004s
```

測試應該注意的幾個事項

1. 不要為了測試而測試
2. 不要強求測試覆蓋率
3. 只針對重要部分做測試，你是來做產品的不是來寫測試的(除非你是QA)
4. 應從單元測試一路往上做到使用者測試

結語

雖然大部分的時候你都沒時間寫測試，但是先寫測試在除錯的時候才會快，相形之下你節省時間更多，而且才能更快確定是哪個模組出錯。

好測試，不寫嗎？

參考資料

- mocha.js:https://mochajs.org
- nightwatch.js:http://nightwatchjs.org
- RIP tdd:https://www.facebook.com/notes/kent-beck/rip-tdd/750840194948847
- cucumber wiki:https://github.com/cucumber/cucumber/wiki
- cucumber.js:https://github.com/cucumber/cucumber-js

前言

當你把程式寫好遠端部署在VPS上，就會碰到一個問題：如何使程式永不中斷的運行？你會發現當你關掉vps的遠端連線後程式就掛了。 你需要pm2，不要再用forever。

安裝 pm2

安裝pm2

```
$ npm install pm2 -g
```

永續執行一隻app

```
$ pm2 start yourapp.js
```

現在即使你關掉了遠端連線他也會持續運轉

- option

如果你想要更新code後會自動restart的話，請使用

```
$ pm2 start yourapp.js --watch
```

他會監看該程式底下的所有目錄，如果有更新會立即restart。

模擬多線程(實驗階段)

javascript本身是單線程的，如果要多線程必須要寫code，聽起來就很麻煩，幸好pm2可以幫我們解決這個問題。

```
$ pm2 start yourapp.js -i 3
```

這樣就會直接啟動三個yourapp.js。

- 注意

因為是多線程，請保證你的session改用redis。

當然效果跟多開yourapp.js是相似的

部署

厭煩了每次都要遠端連上vps用git作部署？pm2可以給你十分良好的部署體驗。

在你的專案根目錄底下新增一個ecosystem.json

```
{
  "apps": [
    {
      "name": "scriptname",//pm2裡顯示的服務名稱
      "script": "start.js"//實際要執行的js檔
    }
  ],
  "deploy": {
    "production": {
      "key": "yousshKey",//ssh key 給pm2遠端連線vps部署用
      "user": "youraccount",//遠端登入的帳號
      "host": "212.83.163.1",//遠端連線的IP
      "ref": "origin/master",//使用哪個git branch
      "repo": "git@github.com:repo.git",//git網址
      "path": "/var/www/production",//部署目錄
      "post-deploy": "sudo npm install && sudo pm2 startOrRestart e
    }
  }
}
```

先下setup指令部署目錄

```
$ pm2 deploy ecosystem.json production setup
```

之後再部署code

```
$ pm2 deploy ecosystem.json production
```

他就會自動安裝ndoe_module跟運行code，之後如果要更新code只需運行

```
$ pm2 deploy ecosystem.json production
```

重開機

有的時候你需要將server重開，但是你不想要一個一個將pm2的程式start，你可以下以下指令：

這會建立pm2的開機程式

```
$ pm2 startup
```

這會儲存pm2現在運行了哪些程式，供下次開機執行

```
$ pm2 save
```

你可以放心重開機了。

結語

pm2還有許多延伸，比如查看記憶體使用狀態，reload。有興趣的人可以參考官網研究一下。

參考資料

- pm2:http://pm2.keymetrics.io

用 Express 和 MongoDB 寫一個 todo list（node版本:4.1.1)

練習一種語言或是 framework 最快的入門方式就是寫一個 todo list 了. 他包含了基本的 C.R.U.D. (新增, 讀取, 更新, 刪除). 這篇文章將用 Node.js 裡最通用的 framework Express 架構 application 和 MongoDB 來儲存資料.

原始檔

https://github.com/y2468101216/node-wiki-gitbook/tree/master/src/todo_list

功能

- 使用 facebook 登入, 用 session 來辨別每一問使用者
- 可以新增, 讀取, 更新, 刪除待辦事項(todo item)

開發環境

開發環境 開始之前請確定你已經安裝了 Node.js, Express 和 MongoDB, 如果沒有可以參考本電子書.

Node.js 套件

本次我們將使用 Node.js, Express, MongoDB, express-generator, ejs 來開發

步驟

用 Express 的 command line 工具幫我們生成一個 project 雛形 預設的 template engine 是 jade, 在這裡我們改用比較平易近人的 ejs.

```
$ express todo_list -e --git
```

(只適用 Mac 使用者)在專案根目錄修改 .gitignore，在最後一行加入

```
#mac style file
.DS_Store
```

Welcome to Express

開啟 express server 然後打開瀏覽器瀏覽 127.0.0.1:3000 就會看到歡迎頁面.

```
$ DEBUG=todo_list npm start
```

Express

Welcome to Express

Project 檔案結構

```
todo_list
|-- bin
|   |-- www
|
|-- node_modules
|   |-- body-parser //負責處理post傳回來的資料
|   |-- cookie-parser //處理cookie
|   |-- debug
|   |-- ejs //view engine
|   |-- express //web server
|   |-- mongodb //db server
|   |-- morgan //紀錄http request log
|   `-- serve-favicon //db server
|
|-- public
|   |-- images
|   |-- javascripts
|   `-- stylesheets
|       |-- style.css
|
|-- routes
|   `-- index.js
|
|-- views
|   |-- index.ejs
|   `-- error.ejs
|
|-- .gitignore
|
|-- app.js
|
`-- package.json
```

- bin - 邏輯檔
- node_modules - 包含所有 project 相關套件.
- public - 包含所有靜態檔案.
- routes - 路由檔.

- views - 包含 action views, partials 還有 layouts.
- app.js - 包含設定, middlewares, 和 routes 的分配.
- package.json - 相關套件的設定檔.

安裝 mongoDB

打開 package.json，在 dependencies 插入兩行

```
  {
   "name": "todo_list",
   "version": "0.0.0",
   "private": true,
   "scripts": {
     "start": "node ./bin/www"
   },
   "dependencies": {
     "body-parser": "~1.13.2",
     "cookie-parser": "~1.3.5",
     "debug": "~2.2.0",
     "ejs": "~2.3.3",
     "express": "~4.13.1",
     "morgan": "~1.6.1",
     "serve-favicon": "~2.3.0",
     "mongodb": "~2.0.0"
   }
 }
```

然後 npm 會自動讀取 package.json

```
npm install
```

就會幫我們 mongoDB 裝好了

安裝測試工具-mocha

我們是好孩子，所以要寫 unit test，詳細的介紹在 NODE_TEST 裡

```
$ npm install -g mocha
```

MongoDB CRUD

我們需要先寫 CRUD 的測試

新增一個 test 目錄並建立一個 dbCRUDTest.js 的檔案，程式碼如下:

```
var dbConnect = require('../bin/dbConnect.js');
var dbConnectTest = new dbConnect();
var crud = require('../bin/dbCRUD.js');
var crudTest = new crud();
var assert = require('assert');

describe('dbTest', function () {
    before('create Test Collection', function (done) {
        console.log('createTestCollection');
        dbConnectTest.connect(function (db) {
            db.createCollection('event', function (err, results) {
                db.close();
                assert.equal(null, err);
                done();
            });
        });
    });

    it('connect Should Be Success', function (done) {
        dbConnectTest.connect(function (db) {
            db.admin().serverInfo(function (err, results) {
                db.close();
                assert.equal(null, err);
                done();
            });
        });
```

```
    });

    it('insert Should Be Success', function (done) {
        dbConnectTest.connect(function (db) {
            crudTest.insert({ userId: '1234', event: 'test' }, db,
                db.close();
                if (err) throw err;
                done();
            });
        });
    });

    it('select Should Not Be 0', function (done) {
        dbConnectTest.connect(function (db) {
            crudTest.select(null, db, function (cursor) {
                cursor.count(function (err, count) {
                    db.close();
                    assert.equal(null, err);
                    assert.notEqual(count, 0);
                    done();
                });

            });
        });
    });

    it('update Should Be Success. But Update Nothing', function (do
        dbConnectTest.connect(function (db) {
            crudTest.update({ _id: 1, event: 'test2', userId: '1234
                db.close();
                assert.equal(null, err);
                assert.equal(0,results.modifiedCount)
                done();
            });
        });
    });

    it('delete Should Be Success. But Delete Nothing', function (do
        dbConnectTest.connect(function (db) {
            crudTest.delete({ _id: 1, userId: '1234' }, db, functio
```

```
                db.close();
                assert.equal(null, err);
                assert.equal(0,results.deletedCount)
                done();
            });
        });
    });

    after('drop Test Collection', function (done) {
        console.log('dropTestCollection');
        dbConnectTest.connect(function (db) {
            db.dropCollection('event', function (err, results) {
                db.close();
                assert.equal(null, err);
                done();
            });
        });

    });
});
```

做測試之前我們需要先把 mongodb 打開

在 Ubuntu 上 MongoDB 開機後便會自動開啟. 在 Mac 上你需要手動輸入下面的指令.

```
$ mongod
```

在bin/下新增一個檔案叫做 dbConnect.js 來設定 MongoDB

```
/**
 * Name:dbConnect.js
 * Purpose:connect mongodb
 * Author:Yun
 * Version:1.0
 * Update:2015-10-15
 */

module.exports = function () {
    var MongoClient = require('mongodb').MongoClient;//mongodb clie
    var url = 'mongodb://localhost:27017/todo_list_test';// mongodb

    this.connect = function (callback) {
        MongoClient.connect(url, function (err, db) {
            if (err) {
                throw err;
            } else {
                callback(db);
            }
        });
    }
}
```

在 bin/ 下新增一個檔案叫做 dbCRUD.js，這是給 todo_list 讀寫資料庫用的。

```
/**
 * Name:dbCRUD.js
 * Purpose:todo_list CRUD
 * Author:Yun
 * Version:1.0
 * Update:2015-10-15
 */

module.exports = function () {

}
```

進行第一次測試，請切換到 todo_list 目錄底下，執行以下指令

```
$ mocha test/dbCRUDTest.js
```

```
dbTest
  ✓ connect (70ms)
  1) select
  2) insert
  3) update
  4) delete


1 passing (263ms)
4 failing

1) dbTest select:
   Uncaught TypeError: crudTest.select is not a function
    at test/dbCRUDTest.js:19:13
    at bin/dbConnect.js:18:5
    at node_modules/mongodb/lib/mongo_client.js:453:11

2) dbTest insert:
   Uncaught TypeError: crudTest.insert is not a function
    at test/dbCRUDTest.js:29:13
    at bin/dbConnect.js:18:5
    at node_modules/mongodb/lib/mongo_client.js:453:11

3) dbTest update:
   Uncaught TypeError: crudTest.update is not a function
    at test/dbCRUDTest.js:39:13
    at bin/dbConnect.js:18:5
    at node_modules/mongodb/lib/mongo_client.js:453:11

4) dbTest delete:
   Uncaught TypeError: crudTest.delete is not a function
    at test/dbCRUDTest.js:49:19
    at bin/dbConnect.js:18:5
    at node_modules/mongodb/lib/mongo_client.js:453:11
```

如上圖所示，你可以注意到雖然 connect 過了，但 是CRUD 沒有過，因為我們還未定義 dbCRUD.js 的 function，讓我們把洞補起來。

修改 dbCRUD.js 如下

```
/**
 * Name:dbCRUD.js
 * Purpose:todo_list CRUD
 * Author:Yun
 * Version:1.0
 * Update:2015-10-15
 */

module.exports = function () {

    this.select = function (findCondition, db, callback) {
        var cursor = db.collection('event').find(findCondition);
        callback(cursor);
    }

    this.insert = function (insertObject, db, callback) {
        db.collection('event').insertOne({event:insertObject.event,
            callback(err, results);
        });
    }

    this.update = function (updateObject, db, callback) {
        var ObjectID = require('mongodb').ObjectID
        var id = new ObjectID(updateObject.id);
        db.collection('event').updateOne({_id:id, userId:updateObje
            callback(err, results);
        });
    }

    this.delete = function (deleteObject, db, callback) {
        var ObjectID = require('mongodb').ObjectID
        var id = new ObjectID(deleteObject.id);
        db.collection('event').deleteOne({_id:id, userId:deleteObje
            callback(err, results);
        });
    }
}
```

執行第二次測試如下

```
$ mocha test/dbCRUDTest.js
```

```
dbTest
  ✓ connect (72ms)
  ✓ select (41ms)
  ✓ insert (46ms)
  ✓ update (44ms)
  ✓ delete (45ms)


5 passing (261ms)
```

這次就全數通過了。

- 備註

這個測試，有兩個不好的地方。

1. connect 跟所有相依在一起了
2. insert 跟 select 相依在一起了

理論上應該要新增一個假的 db instance 去做測試，但是因為作者懶惰的關係，所以決定這樣寫。

修改 index.js 使他可以查詢

我們需要調整 index.js，讓他可以帶查詢的結果

index.js 修改程式碼如下：

```
/**
 * Name:index.js
 * Purpose:show index.html
 * Author:Yun
 * Version:1.0
 * Update:2015-10-20
 */

var express = require('express');
var router = express.Router();

var dbConnect = require('../bin/dbConnect.js');
var dbConn = new dbConnect();

var dbCRUD = require('../bin/dbCRUD.js');
var dbCRUDMethod = new dbCRUD();

/* GET home page. */
router.get('/', function (req, res, next) {
  dbConn.connect(function (db) {
    dbCRUDMethod.select(null, db, function (cursor) {
      var data = [];
      cursor.forEach(function(result){
        data.push(result);
        db.close();
      },function(err){
        if(err) throw err;
        res.render('index', { title: 'Express', cursor: data });
      });

    });
  });

});

module.exports = router;
```

修改 index.ejs 如下:

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
    <ul>
    <% cursor.forEach(function(data){ %>
      <%= data.event %>
    <% }); %>
  </ul>
  </body>
</html>
```

執行:

```
DEBUG=todo_list npm start
```

連http://localhost:3000/即可看到成果

Express

Welcome to Express

這時還沒任何顯示任何資料，因為資料庫裡尚未儲存任何資料。

修改todo_list使其有新增功能

```
<!DOCTYPE html>
<html>

<head>
  <title>
    <%= title %>
  </title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
</head>

<body>
  <p>Welcome to <%= title %></p>
  <form method="post" action="insert">
    <input type="text" value="" placeholder="請輸入代辦事項" />
    <button type="submit" value="">送出</button>
  </form>
  <ul>
    <% cursor.forEach(function(data){ %>
      <%= data.event %>
        <% }); %>
  </ul>
</body>

</html>
```

app.js:

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
```

```
var routes = require('./routes/index');
var users = require('./routes/users');

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

// uncomment after placing your favicon in /public
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', routes);
app.use('/users', users);
app.use('/control/:method', control);

// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});

// error handlers

// development error handler
// will print stacktrace
if (app.get('env') === 'development') {
  app.use(function(err, req, res, next) {
    res.status(err.status || 500);
    res.render('error', {
      message: err.message,
      error: err
    });
  });
```

```
}

// production error handler
// no stacktraces leaked to user
app.use(function(err, req, res, next) {
  res.status(err.status || 500);
  res.render('error', {
    message: err.message,
    error: {}
  });
});


module.exports = app;
```

新增一個control.js檔案在routes底下

```
/**
 * Name:control.js
 * Purpose:update insert delete todo_list
 * Author:Yun
 * Version:1.0
 * Update:2015-10-21
 */

var express = require('express');
var router = express.Router();

/* insert home page. */
router.post('/', function (req, res, next) {
  var Db = require('../bin/DbConnect.js');
  var dbConn = new Db();
  var dbCRUD = require('../bin/dbCRUD.js');
  var dbCRUDControl = new dbCRUD();
  dbConn.connect(function (db) {
    switch (req.query.method) {
      case 'insert':
        dbCRUDControl.insert({ event: req.body.event, userId: 1234
          if (err) throw err;
          db.close();
          res.redirect('/');
        });
        break;
      default:
        res.redirect('/');
        break;
    }
  });
});

module.exports = router;
```

執行:

```
DEBUG=todo_list npm start
```

連 http://localhost:3000/ 即可看到成果

Welcome to Express

請輸入代辦事項 送出

aaaaaa bbbbbb cccc ddddd

完成修改、刪除功能。

index.ejs:

```
<!DOCTYPE html>
<html>

<head>
  <title>
    <%= title %>
  </title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
  <!-- Latest compiled and minified CSS -->
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/boot

  <!-- Optional theme -->
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/boot

  <!-- Latest jquery  -->
```

```
  <script src="https://code.jquery.com/jquery-2.1.4.min.js"></scrip

  <!-- Latest compiled and minified JavaScript -->
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/b
</head>

<body>
  <div class="row">
    <p>Welcome to
      <%= title %>
    </p>
  </div>
  <div class="row">
    <form method="post" action="control?method=insert" class="form-
      <input type="text" name="event" value="" placeholder="請輸入代
      <button type="submit" class="btn btn-primary" value="">送出</
    </form>
  </div>
  <div class="row">
    <table class="table">
      <thead>
        <tr>
          <td>待辦事項</td>
          <td>功能</td>
        </tr>
      </thead>
      <tbody
      <% cursor.forEach(function(data){ %>
        <tr>
          <td>
            <%= data.event %>
          </td>
          <td>
            <button type="button" class="btn btn-default" onclick='
            <button type="button" class="btn btn-default" onclick='
          </td>
        </tr>
        <% }); %>
      </tbody>
    </table>
```

```
    </div>

    <!-- modal for update -->
    <div class="modal fade" id="updateModal">
      <div class="modal-dialog">
        <div class="modal-content">
          <div class="modal-header">
            <button type="button" class="close" data-dismiss="modal"
            <h4 class="modal-title">修改代辦事項</h4>
          </div>
          <form method="post" action="control?method=update">
            <div class="modal-body">
              <input type="text" value="" name="updateImportEventText
              <input type="hidden" value="" name="updateImportEventId
            </div>
            <div class="modal-footer">
              <button type="button" class="btn btn-default" data-dism
              <button type="submit" class="btn btn-warning">Save</but
            </div>
          </form>
        </div>
        <!-- /.modal-content -->
      </div>
      <!-- /.modal-dialog -->
    </div>
    <!-- /.modal -->

    <!-- modal for delete -->
    <div class="modal fade" id="deleteModal">
      <div class="modal-dialog">
        <div class="modal-content">
          <div class="modal-header">
            <button type="button" class="close" data-dismiss="modal"
            <h4 class="modal-title">刪除代辦事項</h4>
          </div>
          <form method="post" action="control?method=delete">
            <div class="modal-body">
              <label>是否刪除</label>
              <input type="hidden" value="" name="deleteImportEventId
            </div>
```

```
          <div class="modal-footer">
            <button type="button" class="btn btn-default" data-dism
            <button type="submit" class="btn btn-warning">是</butto
          </div>
        </form>
      </div>
      <!-- /.modal-content -->
    </div>
    <!-- /.modal-dialog -->
  </div>
  <!-- /.modal -->
</body>

<script>
  function showModal(event, id, method){
    switch (method){
      case 'update':
      $('#updateImportEventId').val(id);
      $('#updateImportEventText').val(event);
      $('#updateModal').modal('show');
      break;
      case 'delete':
      $('#deleteImportEventId').val(id);
      $('#deleteModal').modal('show');
      break;
    }
  }

</script>

</html>
```

control.js:

```
/**
 * Name:control.js
 * Purpose:update insert delete todo_list
 * Author:Yun
```

```
 * Version:1.0
 * Update:2015-10-21
 */

var express = require('express');
var router = express.Router();

/* insert home page. */
router.post('/', function (req, res, next) {
  var Db = require('../bin/DbConnect.js');
  var dbConn = new Db();
  var dbCRUD = require('../bin/dbCRUD.js');
  var dbCRUDControl = new dbCRUD();
  dbConn.connect(function (db) {
    switch (req.query.method) {
      case 'insert':
        dbCRUDControl.insert({ event: req.body.event, userId: 1234
          if (err) throw err;
          db.close();
          res.redirect('/');
        });
        break;
      case 'update':
        dbCRUDControl.update({ event: req.body.updateImportEventTex
          if (err) throw err;
          db.close();
          res.redirect('/');
        });
        break;
      case 'delete':
        dbCRUDControl.delete({ id: req.body.deleteImportEventId, us
          if (err) throw err;
          db.close();
          res.redirect('/');
        });
        break;
      default:
        res.redirect('/');
        break;
    }
```

```
  });
});

module.exports = router;
```

我在 HTML 裡引入了 bootstrap 跟 jquery 以增進使用者體驗跟美化。 並且在 control.js 裡增加了 delete 跟 update 方法。

你可以試著運行看看結果如何。

passport.js 安裝

你可以注意到我們的 userid 都是寫死的，這樣無法區別使用者，所以這邊我們將運用 passport.js 做 facebook 登入。

修改 package.json:

```
{
  "name": "todo_list",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "~1.13.2",
    "cookie-parser": "~1.3.5",
    "debug": "~2.2.0",
    "ejs": "~2.3.3",
    "express": "~4.13.1",
    "morgan": "~1.6.1",
    "serve-favicon": "~2.3.0",
    "mongodb": "~2.0.0",
    "cookie-session":"~2.0.0",
    "passport":"~0.3.0",
    "passport-facebook":"2.0.0"
  }
}
```

passport 需要 session 儲存使用者資訊，這裡我們選擇 cookie-session，還需要擴充模組以支援 facebook 登入

執行:

```
$ npm install
```

- facebook APP 申請

登入 https://developers.facebook.com，點選上方選單的 My apps->Add a New App

點選 Settings，切換到上方的 Advanced 分頁。

往下尋找 Valid OAuth redirect URIs，填寫如圖

拉到最下按 save 就完成了。

- 附註

如果你想要提供給其他人登入的話必須將下圖的電子郵件填寫完畢後

status & Review 中的 status 點選 No 使其變成 Yes

app.js:

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
```

```
var bodyParser = require('body-parser');
var cookieSession = require('cookie-session');

//facebbok login
var passport = require('passport');
var FacebookStrategy = require('passport-facebook').Strategy

var FACEBOOK_APP_ID = "--insert-facebook-app-id-here--"
var FACEBOOK_APP_SECRET = "--insert-facebook-app-secret-here--";

// Passport session setup.
//   To support persistent login sessions, Passport needs to be abl
//   serialize users into and deserialize users out of the session.
//   this will be as simple as storing the user ID when serializing
//   the user by ID when deserializing.  However, since this exampl
//   have a database of user records, the complete Facebook profile
//   and deserialized.
passport.serializeUser(function (user, done) {
  done(null, user);
});

passport.deserializeUser(function (obj, done) {
  done(null, obj);
});


// Use the FacebookStrategy within Passport.
//   Strategies in Passport require a `verify` function, which acce
//   credentials (in this case, an accessToken, refreshToken, and F
//   profile), and invoke a callback with a user object.
passport.use(new FacebookStrategy({
  clientID: FACEBOOK_APP_ID,
  clientSecret: FACEBOOK_APP_SECRET,
  callbackURL: "http://localhost:3000/auth/facebook/callback"
},
  function (accessToken, refreshToken, profile, done) {
    // asynchronous verification, for effect...
    process.nextTick(function () {

      // To keep the example simple, the user's Facebook profile is
```

```
      // represent the logged-in user.  In a typical application, y
      // to associate the Facebook account with a user record in yo
      // and return that user instead.
      return done(null, profile);
    });
  }
  ));

var routes = require('./routes/index');
var users = require('./routes/users');
var control = require('./routes/control');

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

// uncomment after placing your favicon in /public
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(cookieSession({ secret: 'I am your FATHER' }));
app.use(express.static(path.join(__dirname, 'public')));
app.use(passport.initialize());
app.use(passport.session());


app.use('/', routes);
app.use('/users', users);
app.use('/control', control);

// GET /auth/facebook
app.get('/auth/facebook',
  passport.authenticate('facebook'),
  function (req, res) {
    // The request will be redirected to Facebook for authenticatio
    // function will not be called.
```

```
  });

// GET /auth/facebook/callback
app.get('/auth/facebook/callback',
  passport.authenticate('facebook', { failureRedirect: '/' }),
  function (req, res) {
    res.redirect('/');
  });

app.get('/logout', function (req, res) {
  req.logout();
  res.redirect('/');
});

// catch 404 and forward to error handler
app.use(function (req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});

// error handlers

// development error handler
// will print stacktrace
if (app.get('env') === 'development') {
  app.use(function (err, req, res, next) {
    res.status(err.status || 500);
    res.render('error', {
      message: err.message,
      error: err
    });
  });
}

// production error handler
// no stacktraces leaked to user
app.use(function (err, req, res, next) {
  res.status(err.status || 500);
  res.render('error', {
```

```
      message: err.message,
      error: {}
    });
  });

  module.exports = app;
```

index.ejs

```

  <!DOCTYPE html>
  <html>

  <head>
    <title>
      待辦事項
    </title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
    <!-- Latest compiled and minified CSS -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/boot

    <!-- Optional theme -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/boot

    <!-- Latest jquery  -->
    <script src="https://code.jquery.com/jquery-2.1.4.min.js"></scrip

    <!-- Latest compiled and minified JavaScript -->
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/b
  </head>

  <body>
    <% if (name) { %>
    <div class="row">
      <label>Welcome to <%= name %></label><a class="btn btn-warning
    </div>
    <div class="row">
      <form method="post" action="control?method=insert" class="form-
        <input type="text" name="event" value="" placeholder="請輸入待
```

```
      <button type="submit" id="submit" class="btn btn-default" val
    </form>
  </div>
  <% }else{ %>
  <div class="row" id="LoginMessage">
    <div>請先登入FB</div>
    <div>
      <a class="btn btn-primary" href="/auth/facebook">
      Facebook Login
      </a>
    </div>
  </div>
  <% } %>
  <div class="row">
    <table class="table">
      <thead>
        <tr>
          <td>待辦事項</td>
          <td>功能</td>
        </tr>
      </thead>
      <tbody>
        <% if (cursor) { %>
        <% cursor.forEach(function(data){ %>
        <tr>
          <td>
            <%= data.event %>
          </td>
          <td>
            <button type="button" class="btn btn-default" onclick='
            <button type="button" class="btn btn-default" onclick='
          </td>
        </tr>
        <% }); %>
        <% } %>
      </tbody>
    </table>
  </div>

  <!-- modal for update -->
```

```
<div class="modal fade" id="updateModal">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <button type="button" class="close" data-dismiss="modal"
        <h4 class="modal-title">修改代辦事項</h4>
      </div>
      <form method="post" action="control?method=update">
        <div class="modal-body">
          <input type="text" value="" name="updateImportEventText
          <input type="hidden" value="" name="updateImportEventId
        </div>
        <div class="modal-footer">
          <button type="button" class="btn btn-default" data-dism
          <button type="submit" class="btn btn-warning">Save</but
        </div>
      </form>
    </div>
    <!-- /.modal-content -->
  </div>
  <!-- /.modal-dialog -->
</div>
<!-- /.modal -->

<!-- modal for delete -->
<div class="modal fade" id="deleteModal">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <button type="button" class="close" data-dismiss="modal"
        <h4 class="modal-title">刪除代辦事項</h4>
      </div>
      <form method="post" action="control?method=delete">
        <div class="modal-body">
          <label>是否刪除</label>
          <input type="hidden" value="" name="deleteImportEventId
        </div>
        <div class="modal-footer">
          <button type="button" class="btn btn-default" data-dism
          <button type="submit" class="btn btn-warning">是</butto
```

```
          </div>
        </form>
      </div>
      <!-- /.modal-content -->
    </div>
    <!-- /.modal-dialog -->
  </div>
  <!-- /.modal -->
</body>

<script>
  function showModal(event, id, method){
    switch (method){
      case 'update':
      $('#updateImportEventId').val(id);
      $('#updateImportEventText').val(event);
      $('#updateModal').modal('show');
      break;
      case 'delete':
      $('#deleteImportEventId').val(id);
      $('#deleteModal').modal('show');
      break;
    }
  }

</script>

</html>
```

index.js

```
/**
 * Name:index.js
 * Purpose:show index.html
 * Author:Yun
 * Version:1.0
 * Update:2015-10-20
 */
```

```
var express = require('express');
var router = express.Router();

var dbConnect = require('../bin/dbConnect.js');
var dbConn = new dbConnect();

var dbCRUD = require('../bin/dbCRUD.js');
var dbCRUDMethod = new dbCRUD();


/* GET home page. */
router.get('/', function (req, res) {
  dbConn.connect(function (db) {
    if (typeof req.user == 'undefined') {
      res.render('index', { name: false, cursor: false });
    } else {
      dbCRUDMethod.select({userId:req.user.id}, db, function (curso
        var data = [];
        cursor.forEach(function (result) {
          data.push(result);
          db.close();
        }, function (err) {
          if (err) throw err;
          res.render('index', { name: req.user.displayName, cursor

        });

      });
    }

  });

});

module.exports = router;
```

control.js

```
/**
 * Name:control.js
 * Purpose:update insert delete todo_list
 * Author:Yun
 * Version:1.0
 * Update:2015-10-21
 */

var express = require('express');
var router = express.Router();

/* insert home page. */
router.post('/', function (req, res, next) {
  var Db = require('../bin/DbConnect.js');
  var dbConn = new Db();
  var dbCRUD = require('../bin/dbCRUD.js');
  var dbCRUDControl = new dbCRUD();
  if (typeof req.user != 'undefined') {
    dbConn.connect(function (db) {
      switch (req.query.method) {
        case 'insert':
          dbCRUDControl.insert({ event: req.body.event, userId: req
            if (err) throw err;
            db.close();
            res.redirect('/');
          });
          break;
        case 'update':
          dbCRUDControl.update({ event: req.body.updateImportEvent
            if (err) throw err;
            db.close();
            res.redirect('/');
          });
          break;
        case 'delete':
          dbCRUDControl.delete({ id: req.body.deleteImportEventId,
            if (err) throw err;
            db.close();
```

```
              res.redirect('/');
            });
            break;
          default:
            res.redirect('/');
            break;
        }
      });
    } else {
      res.redirect('/');
    }
  });

module.exports = router;
```

最後畫面:

請先登入FB

Facebook Login

待辦事項 功能

Welcome to 陳昱廷 登出

請輸入待辦事項 送出

待辦事項	功能
acdscscsc	修改 刪除

完成

恭喜你，你已經跨出了 Node.js 的第一步，歡迎你加入 Node.js 這個大社群。

附錄

- 伺服器啟用 HTTPS 的教程

精選文章

Express

- `使用 Node.js + Express 從 GET/POST Request 取值 <http://fred-zone.blogspot.com/2012/02/nodejs-express-getpost-request.html>` _ [邀稿中]
- link2 [邀稿中]
- link3 [邀稿中]

參考資源

Node.js 英文書籍

- The Little Book on CoffeeScript,January 2012, Alex MacCaw, O'Reilly Media
- Node for Front-End Developers, January 2012, Garann Means, O'Reilly Media
- Building Hypermedia APIs with HTML5 and Node,2011, Mike Amundsen, O'Reilly Media
- Node Web Development, August 2011, David Herron, PacktPub
- CoffeeScript: Accelerated JavaScript Development,July 2011, Trevor Burnham, Pragmatic Bookshelf
- Getting Started with GEO, CouchDB, and Node.js , July 2011, Mick Thompson, O'Reilly Media
- What Is Node?, July 2011, Brett McLaughlin, O'Reilly Media
- Node: Up and Running, May 2011, Tom Hughes-Croucher, Mike Wilson, O'Reilly Media
- Hands-on Node.js, The Node.js introduction and API reference, May 2011, Pedro Teixeira, LeanPub
- The Node Beginner Book, Manuel Kiessling
- Mastering Node.js,visionmedia

Node.js 中文書籍

- 不一樣的Node.js：用JavaScript打造高效能的前後台網頁程式,2014/05/12, 錢逢祥, 蔡政崇, 林政毅
- Node.js模組參考手冊,2015/04/15,錢逢祥, 蔡政崇, 楊傑文

- 深入淺出Node.js,2014/07/29, 朴靈/編著

Node.js 影音教學

- The Node Sessions: The Best of OSCON 2011, August 2011, O'Reilly Media (Video)
- Tom Hughes-Croucher on Node, March 2011, O'Reilly Media (Video)

Node.js 教學網站

- Node Tuts

Node.js 課程